Soft Comput (2008) 12:3–28
DOI 10.1007/s00500-007-0214-7

FOCUS

# Software development with imperfect information

**Joost Noppen · Pim van den Broek · Mehmet Akşit**

Published online: 19 June 2007


**Abstract** Delivering software systems that fulfill all requirements of the stakeholders is very difficult, if not at all impossible. We consider the problem of coping with imperfect information, like interpreting incomplete requirement specifications or vagueness in decisions, one of the main reasons that makes software design difficult. We define a method for tracing design decisions under imperfect information. To model and compare requirements with estimations, we present fuzzy and stochastic techniques. This approach offers adequate decision support that can deal with imperfect information during software design. The approach is illustrated by a real-world example, based on a storm surge barrier system.

**Keywords** Imperfect information · Fuzzy requirements · Fuzzy estimations · Decision support · Software development

## 1 Introduction

There is now a consensus among the software engineering community that designing even a medium size software system is a complex task (Lethbridge and Laganière 2005). There are many causes for this, such as inherent complexity of the problems to be solved, imprecise, ambiguous and evolving requirements, difficulty of taking the right design decision at the right time, and so on. Although there are many specific reasons why software design projects do not accomplish their original goals, coping with "imperfect information" is a common problem of all projects and possibly the origin of many practical failures. Despite its importance, the "imperfect information" problem has not been studied in the software engineering literature satisfactorily. This article addresses this problem both from practical and theoretical perspectives.

Within the context of the soft computing community, the problem of imperfection has been studied in various forms (Lee et al. 2003; Akşit and Marcelloni 2001a). Although there are slight differences in the terminologies used, *uncertainty* and *impreciseness* are considered as two sub-categories of imperfection. Here, the term uncertainty refers to a transient case, where imperfect information becomes eventually perfect (well known) in due time. On the contrary, imprecise information will always remain imperfect to some degree. Nevertheless, some sort of human justification of imprecise information should be possible.

In software design, although imperfection can be experienced in many ways, in the following we will briefly classify it as *imperfection in contextual information* and *imperfection in design processes*.

- Imperfection in contextual information:
  An important source of contextual information in software development is derived from the related business context and formulated as stakeholders' requirements. In addition, different kinds of contextual information can be collected during the software development process, such as updates of requirements, skills of the available people, available budget and so on.
  Typical types of uncertainties in contextual information are, for example, changes in market demands and

J. Noppen (✉) · P. van den Broek · M. Akşit
TRESE Software Engineering Group, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: noppen@cs.utwente.nl

P. van den Broek
e-mail: pimvdb@cs.utwente.nl

M. Akşit
e-mail: aksit@cs.utwente.nl

definition and introduction of future standards. Although these uncertainties will eventually be known when the software is delivered, during the early stages of software development they are mostly based on estimations. Obviously, wrong estimations will eventually cause wrong product deliveries.

Impreciseness in contextual information generally manifests itself in non-functional requirements. For example, in the requirement "The system must complete the function F in less than T seconds otherwise the user will be annoyed", it is very difficult to precisely define the threshold value of going from "not annoyed" to "annoyed". Instead of instantly being annoyed when the threshold is exceeded, rather the annoyance of the user will gradually increase when the completion time of F increases. One may also experience impreciseness in functional requirements. No matter how formally specified, a requirement such as "The user demands the function F" may be considered imprecise since matching the implementation of F in software and the user's expectation of F (also called customer satisfaction) cannot be, in general, measured precisely. This is because the requirement is based on an interpretation of the user expectation.

- Imperfection in design processes:
  Software engineers have to deal with many kinds of uncertainties, especially in the early phases of software development. For example, software engineers may be forced to decompose a system into a certain modular structure to manage complexity, already in the early phase of software development. On the other hand, it may be preferable to defer this decision to a later phase when the interactions among components are known. This will allow grouping the densely interacting components into the same module for the purpose of improving performance and cohesion. When software is deployed in the customer's environment, the actual satisfaction of the customer may not be determined precisely although it is possible to obtain an imprecise evaluation of the customer on the delivered product.

Unfortunately, although the problem of imperfect information is the source of many practical problems, most current software development methods neglect imperfection during the requirements definition phase completely (Jacobson et al. 1999; Yourdon and Constantine 1979). Rather, the problems that are the result of the imperfection are addressed by iteration and incremental design.

By using an illustrative example, this paper underlines the implication of imperfect information in software design. It is illustrated that ignoring the presence of imperfect information can lead to improper design choices. Since a typical software design process incorporates many cascaded design decisions, the effect of imperfect information can be amplified and lead to deliver products that do not satisfy the stakeholders' requirements.

We propose to extend requirement specifications and alternative evaluations, such that they can represent and utilize the imperfection. In addition, we propose to use architecture evaluation techniques that can reason about this extended model. Imperfection is modeled using probability theory and fuzzy set theory, which are well-known means to capture information that is vague and ambiguous. Additionally, a model will be defined with which it is possible to trace the sequence of design decisions. In case a design turns out to have unsatisfactory quality, the tracing model can be used to reiterate the design while minimizing the adaptation efforts. By using the example case, it is illustrated that modeling imperfect information can lead to designs with better quality.

The remainder of this paper is organized as follows: in Sect. 2 an example case will be presented and the problems will be identified. Section 3 will describe the approach for tracing design decisions and the approach for comparing different impreciseness models. In Sect. 4 we will analyze the example case in case our approach is used. Related work will be described in Sect. 5. In Sect. 6 we will conclude the article.

## 2 An example case and the problem statement

In the following subsections, first we will present an industrial example case. We will then discuss various design alternatives for this example, and explain the problems that are caused by imperfect information. These problems will be addressed in Sect. 3.

### 2.1 An example: remote water sensor

Consider a storm surge barrier designed to protect a moderately populated urban area. The choice of this example is inspired from Tretmans et al. (2001). The barrier has to be closed only in case of absolute necessity; otherwise the cargo transport can be hampered unnecessarily. However, leaving the barrier open during storm situations can result in immediate danger for the population. Since the decision to close the barrier is a complicated task, it has been decided to incorporate a computer-controlled system for this purpose. The control system should make a decision every 10 minutes, based on numerous inputs such as weather forecasts, changes in the water level, tides, etc.

In order to work out this example case, we need to decide on a software design process. For this purpose, in the following, first we will present the functional and non-functional requirements. Second, we will describe the design process. Finally, we will further restrict our scope by making some initial design decisions. We would like to point out that the

techniques proposed in this article are general and the suggested process is introduced for illustration purposes only.

The functional requirements are summarized as follows: The RWS (remote water-level sensor) system should measure the water level of the river and report it periodically to the host computer, which is placed at some other geographic location. The host computer, in turn, should send *control requests* to the RWS. The system architects are requested to analyze the RWS with respect to performance, reliability and cost.

The following non-functional requirements are provided by the stakeholders:

PR1: Client must *on average* receive a water-level reading within 500 ms after sending a control request.
PR2: Client must *at the latest* receive a water-level reading within 500 ms after sending a control request.
PR3: When a failure occurs in the measuring part, the host system must be able to continue operating for 10 seconds.
PR4: The cost of this system must not exceed 225 K euros.

The design process is defined as follows: There will be three design decisions that must be taken in sequence:

(a) The amount of sensors and scheduling of the server has to be decided.
(b) The architecture style has to be selected. This step can be further specialized as the selection of the sensor, server and connection topology.
(c) A subsequent decision on the implementation of the overall architecture has to be made.

Finally, for scoping, we assume that the system architects initially take the following design decisions. First, the RWS is embedded into a system architecture based on the client-server idiom. The remote water-level sensor functionality is encapsulated in a server that serves some number of clients. The RWS server hardware includes an analog to digital converter (ADC) that can read and convert a water-level for one sensor at a time. Requests for water-level readings are queued and fed, one at a time, to the ADC. The ADC measures the water-level of each sensor at the frequency specified by its most recently received control request.

## 2.2 A First look at the expected problems due to Imperfection in design information

We will now consider the possible problems that can occur when current development processes are applied to design problems containing imperfect information. The problems that are identified here are still subject to speculation, since we do not have an improved approach to compare current practices with. However, to have a clear understanding, the problems will be identified first. In Sect. 3 we will define the improved approach.

### *2.2.1 Crisp specification of software quality requirements which are imprecise*

In the Remote Water Sensor example, the quality requirements are provided in a very precise manner. For instance the performance requirement of the system expresses that the maximum response time is 650 ms. However, in practice it can be the case that a value slightly higher than 650 ms (e.g. 651) is still acceptable. The imprecision in the requirements in this case is caused by a certain degree of tolerance of the stakeholder. An alternative that will have resulted in a slightly lower performance is to discard regardless of quality characteristics in other areas, even when the particular combination can be the most suited alternative. Generally, it is very difficult to provide the required precision for every single requirement, also since in specific cases there is tolerance with respect to the desired result. Since software design processes do not address imperfection, impreciseness and uncertainty is mostly completely ignored. Instead of modeling the imperfect information appropriately a best effort choice is made, even when the particular choice can not be justified. In much the same way as described earlier, this choice can lead to discarding appropriate alternatives and result in design adjustments at later stages.

### *2.2.2 Not considering imperfect information in design decisions*

The estimations of quality for the Remote Water Sensor are expressed and used like they are the results of actual measurements. For instance, for the performance of the first option among the alternatives of the communication architecture, an estimation is made of 500 ms. This estimation then is compared with the respective requirement, which was also 500, and therefore the option is evaluated positively. However, since this number is an estimation, it is likely that the actual performance of the eventual system will be different. Due to the fact that the estimation is equal to the allowed average, a small variation can lead to a completely different evaluation of the option. For instance, an estimated average response time of 501 ms will lead to an evaluation of unacceptable quality. Complementary, if all early estimations were 500 ms, but the *final* system can have a response time of 501 ms, the design alternative will have been selected unjustly, even though the variance in the estimation and the actual value is minimal. Due to the crisp character of both the quality requirement specifications and the estimated quality of the design alternatives, a small variation in the requirement specification and/or estimation can have a considerable impact on the final

decision, although in this case both the specification and the estimation are imprecise in nature.

#### *2.2.3 Cascaded errors in design decisions*

Now let us consider the possible design decisions which are taken during the design of the architecture. Like in many practical software projects, a number of design decisions are taken in a sequential manner. In each of these decisions several design alternatives have to be evaluated by comparing them to the requirements. However, since impreciseness in requirements and uncertainty in crisp evaluation are not recognized, the likelihood of selecting a wrong alternative is large. Additionally, the software design process continues after each decision, assuming that all previous decisions have been correct. Only when the current design is no longer satisfactory, will the design process iterate and the design be adjusted. This means that in subsequent decisions the potential error in judgment of the current decision will be cascaded. By ignoring the imperfection the likelihood of ending up with an unsatisfactory design increases, which then leads to corrections by reevaluating the design decisions. While a number of approaches have been proposed for tracing design decisions the relationships between design decisions and the formal motivation for alternative selection are mostly not present. As a result it becomes very difficult to determine the point at which the iteration should be started.

The lack of a formal trace of the design decisions that have been taken makes it impossible to systematically explore the design alternatives that have been identified earlier. For instance, suppose in the Remote Water Sensor in the third design decision, there is no alternative that provides the desired quality. This means that a different system design needs to be considered. Even while the documentation that exists contains the individual quality evaluations, this is not the case for the sequence of the design decisions. Searching for the set of alternatives that offer satisfactory quality therefore becomes an unguided process based on intuition rather than a systematic approach to optimize quality and/or design time. For a design process that consists of a relative small amount of design decisions, such as our example case, this is not necessarily problematic. However, in a typical industrial setting, the amount of design decisions is much larger, which makes it very inefficient to reevaluate every design decision when the design needs to be corrected.

## 3 Addressing imperfection in software design

### 3.1 Introduction

In the previous section we have identified that the assessment of design alternatives is generally inaccurate. Even while most modern design processes address problems that are a result of imperfect information with iteration and incremental design, imperfection should be acknowledged and considered in early stages of software design. To achieve this, the quality evaluation model should be extended such that it is capable of capturing the inherent impreciseness that can occur in quality requirements as well as the uncertainty in quality estimations. In addition, we should also define the means with which evaluations can be made from models containing imperfection.

The evaluation of the quality of a design alternative can be based on the individual quality attributes that have been described in the quality requirements, such as performance or reliability. The overall quality of the alternative is given by a mapping of the set of quality attribute values to an element of a completely ordered set. This mapping can be a very straightforward operation such as a simple addition, or a very complex operation using techniques from Multiple Attribute Decision Making. Comparing the different design alternatives can then be reduced to the comparison of their overall quality. Since the quality requirements often impose restrictions on the allowed values for a specific quality attribute (such as for instance a maximum allowed response time or a maximum cost) it is also possible to directly compare a value with the quality requirement to determine the degree to which the current system satisfies the quality requirements. When this is done, each individual evaluation can be treated uniformly by the mapping function.

### 3.2 Various design alternatives

To illustrate the problems that can occur by not considering imperfection in the early stages of software design, we elaborate on the example case in more detail. In Sect. 3.3 our approach is presented, which is demonstrated with the example case extension.

#### *3.2.1 Alternatives of the sensor server architecture*

When software engineers take design decisions, different solutions are considered and assessed according to their expected quality. To demonstrate the impact that imperfection can have on this, we analyze the decision on the sensor server architecture of the Storm Surge Barrier System. Assume that the architecture contains three kinds of components: water level tasks (independently scheduled units of execution), that are scheduled to run with some period; a shared communication facility task (Comm), that accepts messages from the water level tasks and sends them to a specified client; and the ADC task, which accepts requests from the water level tasks, interfaces with the physical sensors to determine their temperatures, and passes the result back to the requesting water level task. The alternatives for

the server architecture that have been identified by the software engineers lie in the implementation of the ADC and the amount of sensors.

Figure 1 shows three alternatives for the server architecture. The alternative a is based on a single sensor. In this alternative, only one measurement can be performed at a given point in time. During measurement, all requests that arrive will have to wait according to a first come first served principle. We assume that in this option no priority mechanism or scheduling is implemented.

In alternative b the server is connected to multiple sensors and the waterlevel tasks are stacked on to the sensors until all sensors are occupied. Once this completed, new tasks will be added to the set of sensors on a first come first served principle. Again here, we assume that no priority mechanism or scheduling is implemented. This alternative is expected to perform measurements faster on average, than option a, but at a higher cost.

Also in architectural option c, we assume that the server is connected to multiple sensors. In addition, this architecture also contains an intelligent scheduling mechanism based on priority levels of individual tasks so that the most important measurement tasks can be performed as soon as possible.

To compare the three alternatives, the software engineers estimate the average performance, maximum performance, reliability and cost of systems that include one of these alternatives. The estimated behavior is evaluated with respect to the requirements to determine which alternative should be selected. The estimation values and the results of the evaluation are displayed in Table 1.

In Table 1, for each option the estimations are displayed per row. For example, Option 1.1 has an average performance estimation of 400, a maximum performance estimation of 400, a reliability estimation of *infinity* and a cost estimation of 180. The values for $Q_1$, $Q_2$, $Q_3$ and $Q_4$ indicate whether or not the estimations satisfy their respective requirement given in Sect. 2.1. Here, the number "1" indicates the satisfaction condition, whereas the number "0" indicates failure to satisfy the requirement. For example, for Option 1.1 $Q_1$ is 1 because the estimation for average performance is 400, which is better than the requirement of 500 ms for the average performance. Finally the column *Overall Quality* indicates the amount to which the option in its entirety satisfies the quality requirements. For the example, this value is computed by multiplying $Q_1$, $Q_2$, $Q_3$ and $Q_4$.

While the evaluation has lead to two alternatives that can satisfy the quality requirements, the manner in which decision is reached can invalidate the results. Clearly 400 ms is smaller than 500 ms, but since not everything is known about the system at the current point in time, the actual performance is very likely to be different for the completed system. By defining and treating estimations in the same manner as an actual measurement on the finished system would be treated, the inherent imperfection of estimations can invalidate many design decisions at later stages. Similarly the boundaries set by quality requirements can be deceiving. When the average performance is restricted to 500 ms, does this mean that an alternative with a performance of 501 ms is completely unacceptable? In most budgets, for example, there is tolerance with respect to final costs of a software system. How should this be included in the quality requirements, and more specifically in the evaluation of design alternatives?

Note that the presented evaluation approach does not necessarily correspond to the manner in which design alternatives are evaluated in particular design processes. The process presented here is meant as an illustration of a typical software

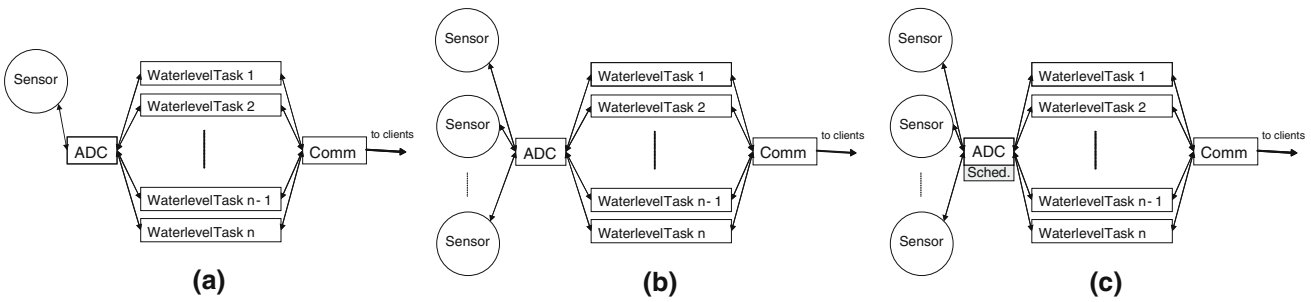



**Fig. 1** Server architecture alternatives: **a** single sensor, **b** multiple sensors, **c** multiple sensors with scheduler

**Table 1** Design Decision 1

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | |
| Opt. 1 | 400 | 400 | $\infty$ | 180 | 1 | 1 | 1 | 1 | 1 |
| Opt. 2 | 350 | 350 | $\infty$ | 190 | 1 | 1 | 1 | 1 | 1 |
| Opt. 3 | 300 | 300 | $\infty$ | 230 | 1 | 1 | 1 | 0 | 0 |

design process. In current software engineering practices, it is quite usual to make estimations as it is carried out in this section (Clements et al. 2004; Kazman et al. 1998), all be it mostly implicitly.

### 3.3 Evaluating design alternatives with respect to requirements

In the previous paragraph we can see that the selection of a solution for a design issue is mostly done by comparing various design alternatives, based on the quality attributes that are considered relevant at the current point in time. However, the quality of a software system can only be determined accurately after a software system has been implemented. Unfortunately, the choice for a design alternative is not taken after the completion of a system, but rather at earlier phases of the design process. The earlier a decision should be taken in the design process, the more difficult it is to estimate the quality behavior of an alternative. In this paragraph we present the first part of our approach with which it is possible to model imperfection in both quality requirements and quality estimations. In addition, we extend the comparison operators that are needed to evaluate design alternatives with respect to the quality requirements.

As has been identified earlier, in both the quality requirements as well as the quality estimations it can be very difficult to determine the precise values required. In our approach we propose that the numeric values that are used in requirements and estimations should be described with probability distributions and fuzzy sets, in addition to normal, "crisp" numbers. By using these models to express the type and nature of imperfect information in the requirements and estimations, it is possible to describe additional knowledge that exists about requirements and estimations, such as tolerance and variance. For instance, in our example the response time of the server depends on the amount of tasks that are waiting in the queue. Rather than estimating the performance of the server with a single number, the performance can more accurately captured by using a probability distribution that describes the arrival rate of tasks in the queue.

In addition to probabilistic imperfection, in our example we also see estimations that are approximations of the actual value. For example, the cost of Option 1.1 is estimated to be 180 k€. However, it is very unlikely that the actual costs will be *exactly* this number. Much rather the costs will be either a slightly lower or slightly higher. We will model this imperfection by means of a *fuzzy set*.

A fuzzy set is a mapping from a domain (cost values in this case) to the numbers in the interval [0, 1]. Each cost value is mapped onto its *degree of membership* in the fuzzy set. In Fig. 2 a fuzzy set is shown that represent a *fuzzy estimation* of the cost of Option 1.1 The cost in this estimation can vary between 160 and 200 k€. It can be seen that the cost value

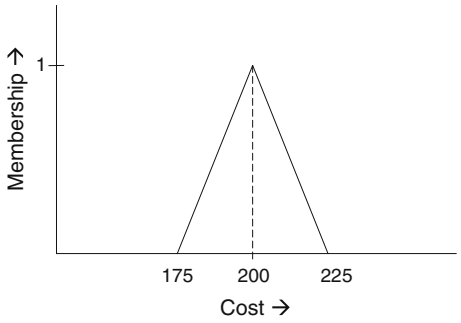



**Fig. 2** Fuzzy set

180 k€ is seen as the most appropriate value, which reflects the "crisp" estimation. Values smaller than 160 k€ and larger than 200 k€ are considered impossible in this estimation, so they have membership value zero. In a similar manner to quality estimations, numerical values in quality requirements can also be represented by fuzzy sets.

While fuzzy sets can take many different shapes, in this paper fuzzy sets are assumed to be triangular fuzzy numbers. A triangular fuzzy number is a fuzzy set on the domain of real numbers whose membership function $\mu$ is given by

$$\begin{aligned} \mu(x) &= 0, && \text{if } x \leq a \\ \mu(x) &= (x-a)(b-a), && \text{if } a \leq x \leq b \\ \mu(x) &= (c-x)(c-b), && \text{if } b \leq x \leq c \\ \mu(x) &= 0, && \text{if } x \geq c \end{aligned}$$

for some real numbers $a, b, c$ with $a \leq b \leq c$, and is denoted by $(a, b, c)$. In this notation the fuzzy number in Fig. 4 is the fuzzy number $(160, 180, 200)$.

In our approach we have identified three different types of imperfect information for quality estimations: imperfection of *probabilistic* nature, imperfection of *fuzzy* nature and imperfection of *fuzzy probabilistic* nature (an imperfection type where it is difficult to exactly specify the parameters of an applicable probability distribution). In addition, we have identified that in quality requirements there can be imperfection of *fuzzy* nature. Since the individual types of imperfect information models are not necessarily of the same type, it is not directly possible to compare estimations and requirements. In our approach we have therefore defined the compa-

**Table 2** Comparison operators reference

| Estimation type | Requirement type | |
|---|---|---|
| | crisp | Fuzzy |
| Crisp | 1, if Est $\leq$ Req, 0 otherwise | Appendix B.4 |
| Probabilistic | Appendix B.1 | Appendix B.5 |
| Fuzzy | Appendix B.2 | Appendix B.6 |
| Fuzzy Probabilistic | Appendix B.3 | Appendix B.7 |

**Table 3** Design Decision 1

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Reliability | Cost | $Q1$ | $Q2$ | $Q3$ | $Q4$ | Overall quality |
| Design Decision 1 | | | | | | | | | |
| Opt. 1 | 400 | 400 | $\infty$ | (160, 180, 200) | 1 | 1 | 1 | 1 | 1 |
| Opt. 2 | 350 | 350 | $\infty$ | (170, 190, 210) | 1 | 1 | 1 | 0.923 | 1 |
| Opt. 3 | 300 | 300 | $\infty$ | (210, 230, 250) | 1 | 1 | 1 | 0 | 0 |

rison operators that are needed. In Table 2 a reference is given to the appendix sections where the definition of comparison operators is given. A more elaborate introduction into probability theory, fuzzy set theory, fuzzy probability theory and how requirements and estimations can be compared using these techniques is given in Appendix A.

In Table 3 we perform the same evaluation as before, but now with *fuzzy estimations* for cost. We use the comparison operators as they are indicated in Table 2.

In Table 3 you can see that with fuzzy estimations the evaluation of the design alternatives changes. Option 1.2 has a lower quality value for the cost evaluation, since the estimation (170, 190, 210) partially exceeds the requirement of 200. By modeling the variance with imperfection models, a better insight in the quality of this design alternative is attained.

## 4 Design history recording using design trees

### 4.1 Introduction

Due to the influence of imperfection in both estimations as well as requirements, and the fact that modern design processes address this by iteration and incremental development, it is very likely that for design decisions alternatives are selected that turn out to be unacceptable. As a result, adjusting designs and redesign of system parts become frequent activities. Since adjusting designs and redesign is a costly operation, searching for a design state where these costs are minimized should ideally be supported by searching algorithms, which can systematically explore the design states. To achieve this, a tracing model is needed, with which it is possible to determine the order in which the design decisions have been addressed. The alternatives that have been considered for each individual design decision as well as the evaluation of the alternatives should be traceable in this model. A model that contains this information can be used to systematically explore the different designs that are available based on the evaluations of individual design solutions.

Additionally, a reasoning algorithm should be defined, which systematically traverses the trace model looking for the best design alternative based on the current knowledge. This should be a configurable algorithm, since the best alternative can depend on managerial interest of the design process, such as minimization of costs, or design for the highest possible quality. This design algorithm should guarantee that space of alternatives is explored in a systematic manner, even when imprecise requirements and estimations are provided. However, we do not aim to achieve automated design. Rather, we aim to provide a set of tools that can support the designer during the design process. This set of tools will consist of an evaluator that can help the designer with design decisions containing imperfect information, and a decision optimizer, which can be used to optimize the selection for the explored design decisions and alternatives.

### 4.2 Cascaded decisions in the example case

In Sect. 3.2 we have used the storm surge barrier example to illustrate that the evaluation of design alternatives, without considering the imperfection that can occur in estimation and requirements, can lead to a faulty assessment. While our imperfection models reduce the likeliness of a faulty assessment, it is still possible to arrive at points where the current design is no longer viable. Since software development processes consists of many design decisions, it becomes very difficult to find the point from which a wrong alternative was chosen. This is even enhanced by the fact that software design processes lack the tracing facilities to capture design decisions and the quality evaluations that were performed on alternatives. In this section we propose the second part of our approach, a trace model called *Design Trees*.

To be able to illustrate our tracing approach, we first extend the storm surge barrier example with two additional design decisions, in much the same manner as Sect. 3. First we decide on which communication architecture to use. Secondly, as a consequence of the second decision, we decide on which extrapolation algorithm we want to use for the intelligent caching mechanism.

#### *4.2.1 Communication architecture*

The first option is indicated with a in Fig. 3. This is a simple and inexpensive client-server architecture, with a single server (RWS Server) and multiple clients. Option b differs from the first option in that it adds a second server to the system architecture. These servers interact with clients as a

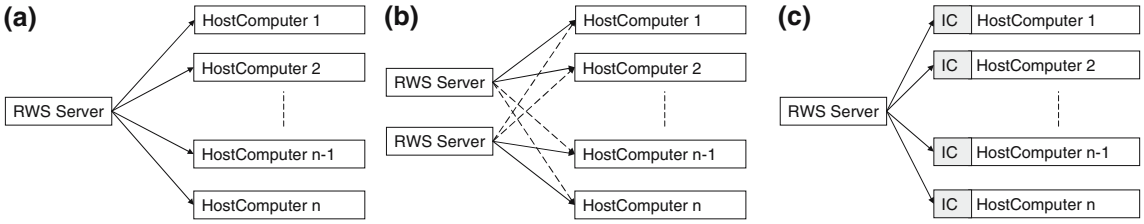



**Fig. 3** Communication architectures: **a** single server, **b** redundant server, **c** intelligent caching

"primary" server (indicated by the solid lines between servers and clients) or as a "backup" server (indicated by the dashed lines). Every client will automatically switch to their specified backup if they detect that the main server is down (because it has failed to send requests for a prescribed period of time). Option c extends option 1 by a "wrapper" that intercedes between the client and the server. This wrapper is an "intelligent cache", shown as IC in the figure. The cache intercepts periodic water level updates from the server to the client, builds a history of these updates, and then passes each update to the client. When the server is interrupted, the cache synthesizes updates for the client. The cache is considered as *intelligent* because the updates it provides take advantage of historical water level trends to extrapolate plausible values into the future. This intelligence may be nothing more than linear extrapolation or it can be a sophisticated model that analyzes changes in temperature trends, or takes advantage of domain-specific knowledge on how water levels rise and fall. Obviously, the synthesized updates of the cache will become less meaningful over time. In Table 4 you can find the evaluations of the design alternatives of the second design decision, after choosing the second option at the first decision.

The performance of the first two options on average is identical, since they both use the same server (but the second option has a redundant server). The third option is obviously slower, since the intelligent caching needs to be updated. The maximum performance is indefinitely long for the first option since in case the server fails, there will be no reply. The second option will wait for a timeout of the first server before the second server sends the measurement. The third option has a maximum performance identical to the average, since the cache can provide "measurements" any time the server fails. The reliability for the first option is 0, since in case of a server failing, the system is not able to continue running. For the second option this is infinite, since in case of a server failing the system can continue operating normally. For the third option, the reliability depends on the time the intelligent cache is able to provide sensible extrapolated values. Finally the cost for the multiple servers and intelligent caching is estimated higher than the single server solution.

### *4.2.2 Alternatives of the intelligent cache*

The final design decision to be made is with respect to the type of intelligent cache that will be used. In this example case three different cache implementations are considered: *Linear Extrapolation, Trend Extrapolation* and *Domain Analysis Extrapolation*.

In linear extrapolation, we assume that only the values that have occurred recently from the sensors are considered. In this case, the cache does not need to keep track of a large number of measurement values. However, a linear extrapolation cannot be used over extended periods of time, since it does not keep track of the periodical behavior of rivers for instance caused by rainfall or temperature changes.

The trend extrapolation cache analyses the trends that have occurred in the available measurements, and tries to extrapolate multiple values according to this trend. For this type of extrapolation a larger set of values needs to be cached in order to make a reliable trend analysis (the actual amount of data depends on the kind of trend analysis). In addition to the amount of data required, the computational complexity also increases, since the trend analysis must be performed as well as the extrapolation.

The trend analysis cache includes specific knowledge on how water levels change over time. This can for instance

**Table 4** Design Decision 2

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 2 | | | | | | | | | |
| Opt. 2.1 | 400 | $\infty$ | 0 | 190 | 1 | 0 | 0 | 1 | 0 |
| Opt. 2.2 | 400 | 650 | $\infty$ | 200 | 1 | 1 | 1 | 1 | 1 |
| Opt. 2.3 | 450 | 450 | 13 | 205 | 1 | 1 | 1 | 1 | 1 |

**Table 5** Design Decision 3

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 3 | | | | | | | | | |
| Opt. 3.1 | 510 | 510 | 9.5 | 205 | 0 | 1 | 1 | 1 | 0 |
| Opt. 3.2 | 500 | 500 | 10 | 225 | 1 | 1 | 1 | 1 | 1 |
| Opt. 3.3 | 850 | 850 | 12 | 300 | 0 | 0 | 1 | 0 | 0 |

be knowledge on seasonal swings in water levels caused by precipitation or temperature levels. Together with a trend analysis based on recent data from the sensors this domain knowledge can be used to perform an informed extrapolation. This should result in the possibility to provide credible extrapolations for a prolonged period of time.

The three alternatives have been evaluated and the results of the quality estimations and evaluation are given in Table 5.

The performance for the first option is estimated at 510, which is slightly higher than the second option. This is due to the fact that a linear extrapolation always needs to consider the newest value that has been measured to determine the linearity. The trend extrapolation does not necessarily need to do this. The third option always needs to consider a complex mathematical model of the environment variables, which makes the performance much lower. The reliability for the linear extrapolation is somewhat lower than the trend extrapolation, since it has a simpler means of extrapolation of sensor readings. The third option is obviously superior in this field. Finally the cost of each option increases as the complexity of the extrapolation algorithm increases.

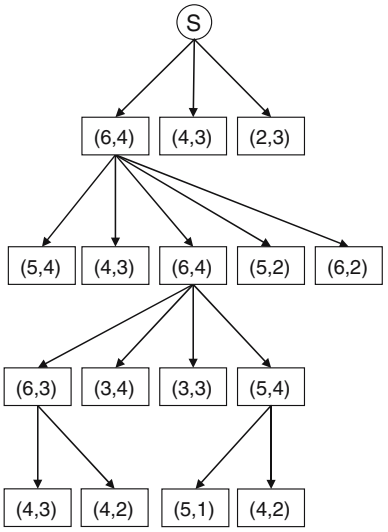



**Fig. 4** Design tree

### 4.3 Design trees

The design of software can be seen as a process of steps,[1] in which customer requirements are transformed into a software system that incorporates these requirements. In each step one of the remaining design issues is resolved. When the software engineer arrives at a point where a satisfactory system design is no longer possible, he has to roll back to a previous, more promising design state. To enable software engineers to examine the previous states systematically, we propose that the design decisions are traced using a *Design Tree*. A design tree is a tree that contains all the design decisions that have been made, their sequence, and the alternatives that have been considered. A sample design tree is depicted in Fig. 4. With this design tree model, software design can be seen as a search problem within a search space, which is comprised of the alternative system designs that theoretically could be considered. This search space is a tree structure that is comprised of all possible alternative system designs, and is called the *principle design tree*.

In the principal design tree leave nodes are the *completed designs* and all other nodes are *partial designs*. Partial designs are designs, which have at least one *design issue* to be resolved, before the design phase is completed. One of the design issues is chosen to be the *principal design issue*, which is the design issue that needs to be resolved first. The principle design issue can be resolved by a number of functionally equivalent *alternative solutions*. These alternative solutions determine the (partial) designs which are the children of the current partial design.

In the design process, the principle design tree is explored until a satisfactory design is found (a leave is reached that satisfies the requirements). The current state of the design process is given by a *design tree*, which is equal to the current part of the principle design tree, which has been explored thus far. At each step in the design process a node of the design tree is expanded, i.e. (a subset of) its children in the principle design tree are added to the design tree. Since the principle design tree is usually too big to explore completely, the design tree is only expanded until a design is found of acceptable quality.

[1] Most practical methods define a set of sequential steps. In general, some parallelism among different steps may be possible.

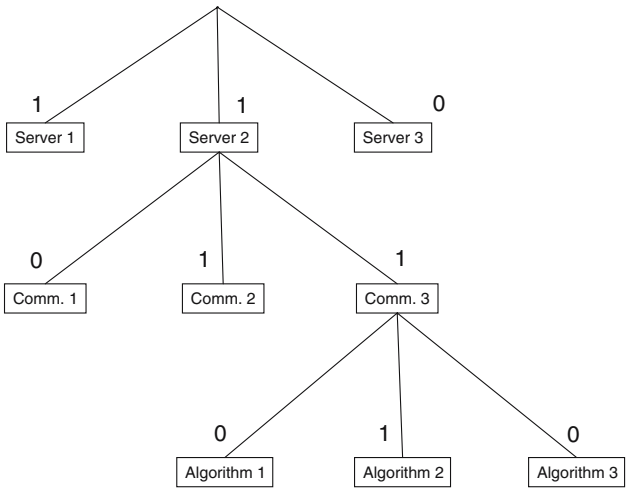



**Fig. 5** Design decisions for the Storm Surge Barrier

In Fig. 5 a design tree is depicted, that represents the three design decisions that have been taken for the Storm Surge Barrier, as well as their overall quality evaluation.

### 4.4 Optimization strategies for design processes

In Sect. 3 it was explained, that the selection of design alternatives is done by comparing the expectations of the relevant quality attributes. In a design tree, the step of choosing a particular design alternative is represented by expanding the node that corresponds to a design that includes this alternative. However, the tracing capabilities of the design tree enable the software engineer to continue from any leaf node, and not only child nodes of the current design state. This makes it possible to switch to previous design states, when the current design state no longer offers acceptable quality expectations. As a result the design tree can be used to determine at which point an iteration cycle should start.

To determine in a systematic manner which node should be expanded, all leaf nodes are sorted based on various attributes, such as quality expectations, depth in the tree, etc. The selection of the node of the design tree to be expanded is therefore determined by the way the nodes are ordered, or the so called *design strategy*. Note that more than one design strategy can exist, for instance a strategy that searches for the best possible system or a strategy that searches for a low-cost system of acceptable quality. The preference of one strategy over the others is based on managerial motives such as minimization of costs, or time to market. We will present three design strategies that can be applied during the design process. To ensure correct results it is assumed that all quality estimations are made in an optimistic manner, meaning that the estimated quality should always be greater than or equal to the actual quality that can be achieved.

One of the most time-consuming operations is to traverse the design tree to find the new node to expand. For this purpose a list-based storage-and-retrieval structure will be defined to be able to access the nodes easily. A list $L$ contains all the leaves of the design tree. The nodes are ranked based on the design strategy in such a way that the node to be selected is the first node of $L$. Whenever a node is expanded, this node is replaced in the list by all its identified child nodes. After this operation the list is ordered again. Note that the design strategies themselves are variants of the branch-and-bound searching algorithm, and are in particular variants of the well-known A*-search algorithm, which is for instance described in Russel and Norvig (1995).

A general algorithm for this process can is depicted in Fig. 6.

In this algorithm the function *Sort* rearranges the list $L$ such that it becomes an ordered list. The strategies are implemented in the Sort-function. This means the design strategies only differ in the comparison criterion for two nodes. Below we will describe three different design strategies that can be applied in the design process.

The first strategy, aimed at finding the optimal design, uses a comparison based on only the (optimistic) quality estimation. The nodes are ordered based on their individual quality estimations, with the node with the highest estimation ordered on top. This strategy guarantees to find the best design possible; however, due to the need to explore the entire principle design tree, this strategy will take a very long time.

A second strategy can therefore be directed at minimizing the time of the development process, and therefore tries to find a design, *any* design, as soon as possible. Since the depth of a node in the design tree indicates how many design issues have been resolved, a deeper node is closer to a completed design. Therefore we can define a fast strategy by always choosing the lowest leaf node in the tree, and in case two or more nodes are at the same depth, the node with the highest quality estimation is taken. To achieve this strategy, in addition to the quality estimation at each node, also the depth of the node in the tree is needed. Therefore the value of a node

```
Design
{
      List L = { Root Node };
      Node N = First element of L;

      While (N is not a completed design)
      {
            Remove N from L;
            Add Children of N to L;
            Sort(L);
            N = First element of L;
      }
      Return N;
}
```

**Fig. 6** General design algorithm

in the design tree is represented by a 2-tuple of type:

$$(Depth, Quality\ Value)$$

The first element of the tuple is the depth in the tree and the second element is the number that represents the actual quality estimation of the node. Now it is possible to compare two nodes based on the standard comparison operator for tuples:

$$(n_1, m_1) > (n_2, m_2) \Leftrightarrow (n_1 > n_2) \vee ((n_1 = n_2) \wedge (m_1 > m_2))$$

Note that this strategy aims to find a design as soon as possible and disregards any quality constraints, which means there is no guarantee that the system will satisfy any quality requirements.

The third design strategy therefore aims at offering a trade-off between quality of the system and the performance of the design strategy. In this strategy the node that is lowest in the tree and still satisfies the requirements is selected. This strategy therefore is aimed at finding a design that has sufficient quality as soon as possible. For this, another criterion is needed, a Boolean that indicates whether the node satisfies the quality constraint. The value of a node in the design tree is represented by a 3-tuple of type

$$(Boolean, Depth, Quality\ Value).$$

The first element is the truth value of the statement "The quality estimation of the system satisfies the quality constraint". The second element is the depth of the node from the root of the tree. The third element is the actual quality estimation of the node. The final design that is found by this strategy satisfies the quality constraint (if such a design exists), but it need not be the design with the highest quality. This strategy will find an acceptable system rather than the best system, which is the result of the first strategy. Using the standard comparison operator for 3-tuples however, the strategy needs less time, and therefore is more desirable in software system design.

It is important to note that the algorithm described in this paragraph is not aimed at automating design. This can, for instance, be seen from the fact that the child nodes (being the alternative solutions to a particular design decision) need to be identified by the engineer, as well as the actual quality evaluation of each alternative. Much rather, the algorithm defines a structured way of exploring the available alternatives, and provides decision support based on the available information. We will demonstrate the various results of the design strategies in Sect. 5.

## 5 Analysis of the approach using the example case

In this section the proposed approach will be analyzed with respect to the example case of Sect. 2. First, the example case will be reevaluated when the estimations contain uncertain information. Second, the requirements will also contain imprecision.

### 5.1 Analysis when considering uncertainty in quality estimations

First, we will introduce uncertainty into the estimations that are made on the expected quality of the final system. The difference between the estimated quality and the eventual quality of the system can have a considerable impact on the design process.

#### *5.1.1 Probabilistic estimations of performance*

Suppose the performance estimations are based on probability models rather than crisp numbers. This can represent the fact that at any given time the response time of the system depends on the amount of requests that are waiting in the request queue. In our case we will assume that the exponential distribution, given by $f(x) = \lambda * e^{-\lambda x}$, is used to model the expected response times. Its mathematical expectation value is $1/\lambda$ (a more detailed description of the use of probability density functions can be found in Appendix A). We will now reevaluate the results from the table by interpreting the estimated values for both the average and maximum response time with an exponential distribution.

In Table 6 the performance estimation is done by use of a exponential probability distribution, which means that for every single response time a non-zero probability of occurrence exists (also for response times that are larger than the required maximum). In the table this is shown by making the maximum estimated response time infinitely large (indicated by $\infty$). The value for $Q_2$ in the table for probability distributions is defined as the fraction of the response times that adhere to the requirements (see Appendix B for details). Therefore the value in the table for $Q_2$ represents the fraction of the responses which are less than 650 ms. The overall result does not change significantly Table 4, besides that the resulting values are somewhat lower.

#### *5.1.2 Fuzzy estimations of reliability and costs*

In addition to making more refined estimations using probability distributions it is also possible to refine estimations using fuzzy sets. For instance, in case of costs or reliability it can be impossible to make an exact estimation, but a global estimation might be possible. For instance, instead of a total cost of 200 k€ the best specification that can be given is

**Table 6** Decision evaluation with probabilistic performance estimations

| | Performance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\lambda$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | 1/400 | 400 | $\infty$ | $\infty$ | 180 | 1 | 0.803 | 1 | 1 | 0.803 |
| Opt. 1.2 | 1/350 | 350 | $\infty$ | $\infty$ | 190 | 1 | 0.844 | 1 | 1 | 0.844 |
| Opt. 1.3 | 1/300 | 300 | $\infty$ | $\infty$ | 230 | 1 | 0.885 | 1 | 0 | 0 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | 1/400 | 400 | $\infty$ | 0 | 190 | 1 | 0.803 | 0 | 1 | 0 |
| Opt. 2.2 | 1/400 | 400 | $\infty$ | $\infty$ | 200 | 1 | 0.803 | 1 | 1 | 0.803 |
| Opt. 2.3 | 1/450 | 450 | $\infty$ | 13 | 205 | 1 | 0.764 | 1 | 1 | 0.764 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | 1/510 | 510 | $\infty$ | 9.5 | 205 | 0 | 0.720 | 0 | 1 | 0 |
| Opt. 3.2 | 1/500 | 500 | $\infty$ | 10 | 225 | 1 | 0.727 | 1 | 1 | 0.727 |
| Opt. 3.3 | 1/850 | 850 | $\infty$ | 12 | 300 | 0 | 0.534 | 1 | 0 | 0 |

**Table 7** Decision evaluation with fuzzy estimations for reliability and costs

| | Performance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\lambda$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | 1/400 | 400 | $\infty$ | $\infty$ | (155,180,205) | 1 | 0.803 | 1 | 1 | 0.803 |
| Opt. 1.2 | 1/350 | 350 | $\infty$ | $\infty$ | (165,190,215) | 1 | 0.844 | 1 | 1 | 0.844 |
| Opt. 1.3 | 1/300 | 300 | $\infty$ | $\infty$ | (205,230,255) | 1 | 0.885 | 1 | 0.239 | 0.212 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | 1/400 | 400 | $\infty$ | 0 | (165,190,215) | 1 | 0.803 | 0 | 1 | 0 |
| Opt. 2.2 | 1/400 | 400 | $\infty$ | $\infty$ | (175,200,225) | 1 | 0.803 | 1 | 1 | 0.803 |
| Opt. 2.3 | 1/450 | 450 | $\infty$ | (12,13,14) | (180,205,230) | 1 | 0.764 | 1 | 0.983 | 0.751 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | 1/510 | 510 | $\infty$ | (8.5,9.5,10.5) | (180,205,230) | 0 | 0.720 | 0.076 | 0.983 | 0 |
| Opt. 3.2 | 1/500 | 500 | $\infty$ | (9,10,11) | (200,225,250) | 1 | 0.727 | 0.5 | 0.5 | 0.182 |
| Opt. 3.3 | 1/850 | 850 | $\infty$ | (11,12,13) | (275,300,325) | 0 | 0.534 | 1 | 1 | 0 |

*approximately* 200 k€. For our case we assume that for both the reliability and the cost such impreciseness occurs. In the table below the reliability and cost attributes are expressed and evaluated using triangular fuzzy numbers (see Sect. 3).

As can be seen in Table 7, a small variation in the cost and reliability estimation using fuzzy set modeling results in a substantially different overall quality estimation. Since the crisp estimations were exactly equal to the requirement the alternatives were completely acceptable. However, with the fuzzy estimation, half the estimation is larger than the required value, which leads to a much lower evaluation. As a result option 2.2 is now rated higher than option 2.3, and option 3.2 receives a very low quality evaluation compared with the value in the previous table.

#### *5.1.3 Fuzzy probabilistic estimations for performance*

In addition to the probabilistic estimation of performance, it can be the case that the exact probability distribution is not known. In this case fuzzy probability distributions can be used (Buckley 2003). Let us assume that for our example an uncertain estimation of performance is done with an exponential fuzzy probability distribution. This means that the parameter $\lambda$ in $f(x) = \lambda * e^{-\lambda x}$ is replaced by a fuzzy number, denoted by $\lambda_f$. In our example, $\lambda$ is replaced by a triangular fuzzy number $(\lambda - 0.0005, \lambda, \lambda + 0.0005)$. This will lead to the following evaluation results:

In Table 8 $f_x$ stands for a fuzzy number with the highest degree of membership at $x$. This is non-triangular fuzzy

**Table 8** Decision evaluation with fuzzy probabilistic estimations for performance

| | Performance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\lambda_f$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | ∞ | ∞ | (155,180,205) | 1 | 0.798 | 1 | 1 | 0.798 |
| Opt. 1.2 | (1/350−1/2000, 1/350, 1/350+1/2000) | $f_{350}$ | ∞ | ∞ | (165,190,215) | 1 | 0.840 | 1 | 1 | 0.840 |
| Opt. 1.3 | (1/300−1/2000, 1/300, 1/300+1/2000) | $f_{300}$ | ∞ | ∞ | (205,230,255) | 1 | 0.882 | 1 | 0.239 | 0.210 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | ∞ | 0 | (165,190,215) | 1 | 0.798 | 0 | 1 | 0 |
| Opt. 2.2 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | ∞ | ∞ | (175,200,225) | 1 | 0.798 | 1 | 1 | 0.798 |
| Opt. 2.3 | (1/450−1/2000, 1/450, 1/450+1/2000) | $f_{450}$ | ∞ | (12,13,14) | (180,205,230) | 0.872 | 0.758 | 1 | 0.983 | 0.650 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | (1/510−1/2000, 1/510, 1/510+1/2000) | $f_{510}$ | ∞ | (8.5,9.5,10.5) | (180,205,230) | 0.301 | 0.713 | 0.076 | 0.983 | 0.016 |
| Opt. 3.2 | (1/500−1/2000, 1/500, 1/500+1/2000) | $f_{500}$ | ∞ | (9,10,11) | (200,225,250) | 0.438 | 0.720 | 0.5 | 0.5 | 0.079 |
| Opt. 3.3 | (1/850−1/2000, 1/850, 1/850+1/2000) | $f_{850}$ | ∞ | (11,12,13) | (275,300,325) | 0 | 0.522 | 1 | 1 | 0 |

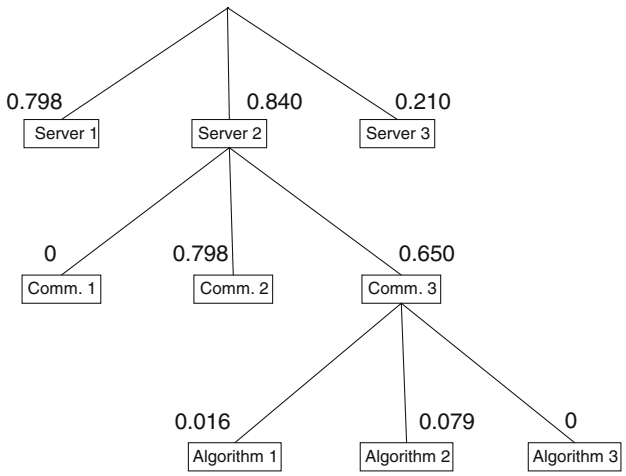



**Fig. 7** Design tree with imprecise estimations

number, which is the fuzzy average of the fuzzy probability distribution. For more information, see Appendix B.

From the table it can be seen that a fuzzy probabilistic estimation for reliability severely influences the degree of fulfillment for individual quality attributes. Option 3.2 even has an evaluation of 0.079, while in the crisp evaluation it had an evaluation of 1, a difference of 0.921. Clearly this is caused by the fact that all the estimations were very close or even equal to the requirements, which means that a slight variation can have a considerable impact. The results in the table are depicted in a design tree in Fig. 7.

From this design tree it can be seen that the evaluation of the alternatives during the first two design decisions has been considerably optimistic in the crisp case. When the uncertainty in the estimations is modeled explicitly using probabilistic and fuzzy set models the alternatives have a much lower quality evaluation than in the crisp case.

## 5.2 Analysis when considering imprecision and uncertainty

As in the estimations, impreciseness can also manifest itself in the requirements. However, in the case of requirements it represents a certain tolerance with respect to a boundary to which the system should adhere. For the example case we will assume that the boundaries of $PR_1$, $PR_2$, $PR_3$ and $PR_4$ are relaxed by allowing a certain amount of tolerance, which is represented by fuzzy numbers.

$PR_1 : (400, 500, 600)$
$PR_2 : (550, 650, 750)$
$PR_3 : (8, 10, 12)$
$PR_4 : (215, 225, 235)$

When we evaluate these fuzzy requirement specifications with the crisp estimations from the initial table, this leads to the result in Table 9.

It can be seen that the specification of impreciseness in requirements influences the evaluation of the design alternatives when the estimations are within the tolerance range of the requirements. In the third design decision the evaluation of the first alternative changes from 0 to 0.675 since 9.5 is inside the tolerance range of the fuzzy requirement. This means that for the third design decision two options can be considered instead of only one for the crisp case.

### *5.2.1 Probabilistic estimations of performance*

As in Sect. 4.1.1 we will now reevaluate the results from the table by interpreting the estimated values for performance as *average* response time for systems with an exponential performance distribution.

**Table 9** Evaluating fuzzy requirements with crisp estimations

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | |
| Opt. 1.1 | 400 | 400 | $\infty$ | 180 | 1 | 1 | 1 | 1 | 1 |
| Opt. 1.2 | 350 | 350 | $\infty$ | 190 | 1 | 1 | 1 | 1 | 1 |
| Opt. 1.3 | 300 | 300 | $\infty$ | 230 | 1 | 1 | 1 | 0.5 | 0.5 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | |
| Opt. 2.1 | 400 | $\infty$ | 0 | 190 | 1 | 0 | 0 | 1 | 0 |
| Opt. 2.2 | 400 | 650 | $\infty$ | 200 | 1 | 1 | 1 | 1 | 1 |
| Opt. 2.3 | 450 | 450 | 13 | 205 | 1 | 1 | 1 | 1 | 1 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | |
| Opt. 3.1 | 510 | 510 | 9.5 | 205 | 0.9 | 1 | 0.75 | 1 | 0.675 |
| Opt. 3.2 | 500 | 500 | 10 | 225 | 1 | 1 | 1 | 1 | 1 |
| Opt. 3.3 | 850 | 850 | 12 | 300 | 0 | 0 | 1 | 0 | 0 |

**Table 10** Evaluating fuzzy requirements with probabilistic performance estimations

| | Performance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\lambda$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | 1/400 | 400 | $\infty$ | $\infty$ | 180 | 1 | 0.826 | 1 | 1 | 0.826 |
| Opt. 1.2 | 1/350 | 350 | $\infty$ | $\infty$ | 190 | 1 | 0.864 | 1 | 1 | 0.864 |
| Opt. 1.3 | 1/300 | 300 | $\infty$ | $\infty$ | 230 | 1 | 0.903 | 1 | 0.5 | 0.452 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | 1/400 | 400 | $\infty$ | 0 | 190 | 1 | 0.826 | 0 | 1 | 0 |
| Opt. 2.2 | 1/400 | 400 | $\infty$ | $\infty$ | 200 | 1 | 0.826 | 1 | 1 | 0.826 |
| Opt. 2.3 | 1/450 | 450 | $\infty$ | 13 | 205 | 1 | 0.788 | 1 | 1 | 0.788 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | 1/510 | 510 | $\infty$ | 9.5 | 205 | 0.9 | 0.746 | 0.75 | 1 | 0.504 |
| Opt. 3.2 | 1/500 | 500 | $\infty$ | 10 | 225 | 1 | 0.753 | 1 | 1 | 0.753 |
| Opt. 3.3 | 1/850 | 850 | $\infty$ | 12 | 300 | 0 | 0.561 | 1 | 0 | 0 |

In Table 10 it can be seen that the overall evaluations are somewhat lower. Additionally, option 3.1 and 3.2 do not differ much with respect to their overall evaluation, where in the crisp case option 3.1 was evaluated with a 0.

#### *5.2.2 Fuzzy estimations of reliability and costs*

As in Sect. 4.1.2 the estimations of reliability and costs are triangular fuzzy numbers. The results are summarized in Table 11.

When evaluating the fuzzy estimations with the fuzzy requirements, it is interesting to see that option 3.1 and 3.2 have become almost equal with respect to their overall evaluation. This is logical since the estimations for both options differed only slightly.

#### *5.2.3 Fuzzy probabilistic estimations of performance*

As in Sect. 4.1.3 the fuzzy probability parameter $\lambda$ is replaced by $(\lambda - 0.0005, \lambda, \lambda + 0.0005)$.

In Table 12 the most obvious changes remain option 1.1 with an evaluation larger than 0, and options 3.1 and 3.2 with almost equal evaluation.

Now that all the estimations and requirements have been modeled using impreciseness models the following design tree can be depicted in Fig. 8.

The selection of the nodes to be expanded in the crisp case would have lead to a node with an actual quality 0.495. In the crisp case, this node was the only one with acceptable quality. However, when imperfection is included it turns out that option 2.2 is actually more promising that option 2.3. Additionally, options 3.1 and 3.2 are very close in their evaluation,

**Table 11** Evaluating fuzzy requirements with fuzzy estimations for reliability and costs

| | Performance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\lambda$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | 1/400 | 400 | $\infty$ | $\infty$ | (155,180,205) | 1 | 0.826 | 1 | 1 | 0.826 |
| Opt. 1.2 | 1/350 | 350 | $\infty$ | $\infty$ | (165,190,215) | 1 | 0.864 | 1 | 1 | 0.864 |
| Opt. 1.3 | 1/300 | 300 | $\infty$ | $\infty$ | (205,230,255) | 1 | 0.903 | 1 | 0.257 | 0.232 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | 1/400 | 400 | $\infty$ | 0 | (165,190,215) | 1 | 0.826 | 0 | 1 | 0 |
| Opt. 2.2 | 1/400 | 400 | $\infty$ | $\infty$ | (175,200,225) | 1 | 0.826 | 1 | 1 | 0.826 |
| Opt. 2.3 | 1/450 | 450 | $\infty$ | (12,13,14) | (180,205,230) | 1 | 0.788 | 1 | 1 | 0.788 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | 1/510 | 510 | $\infty$ | (8.5,9.5,10.5) | (180,205,230) | 0.9 | 0.746 | 0.725 | 1 | 0.487 |
| Opt. 3.2 | 1/500 | 500 | $\infty$ | (9,10,11) | (200,225,250) | 1 | 0.753 | 1 | 0.7 | 0.527 |
| Opt. 3.3 | 1/850 | 850 | $\infty$ | (11,12,13) | (275,300,325) | 0 | 0.561 | 1 | 0 | 0 |

**Table 12** Evaluating fuzzy requirements with fuzzy probabilistic performance estimations

| | Performance | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\lambda_f$ | Avg. | Max. | Reliability | Cost | $Q_1$ | $Q_2$ | $Q_3$ | $Q_4$ | Overall quality |
| Design Decision 1 | | | | | | | | | | |
| Opt. 1.1 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | $\infty$ | $\infty$ | (155,180,205) | 1 | 0.908 | 1 | 1 | 0.908 |
| Opt. 1.2 | (1/350−1/2000, 1/350, 1/350+1/2000) | $f_{350}$ | $\infty$ | $\infty$ | (165,190,215) | 1 | 0.933 | 1 | 1 | 0.933 |
| Opt. 1.3 | (1/300−1/2000, 1/300, 1/300+1/2000) | $f_{300}$ | $\infty$ | $\infty$ | (205,230,255) | 1 | 0.956 | 1 | 0.257 | 0.246 |
| Design Decision 2 after choosing option 1.2 | | | | | | | | | | |
| Opt. 2.1 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | $\infty$ | 0 | (165,190,215) | 1 | 0.908 | 0 | 1 | 0 |
| Opt. 2.2 | (1/400−1/2000, 1/400, 1/400+1/2000) | $f_{400}$ | $\infty$ | $\infty$ | (175,200,225) | 1 | 0.908 | 1 | 1 | 0.908 |
| Opt. 2.3 | (1/450−1/2000, 1/450, 1/450+1/2000) | $f_{450}$ | $\infty$ | (12,13,14) | (180,205,230) | 1 | 0.881 | 1 | 1 | 0.881 |
| Design Decision 3 after choosing option 2.3 | | | | | | | | | | |
| Opt. 3.1 | (1/510−1/2000, 1/510, 1/510+1/2000) | $f_{510}$ | $\infty$ | (8.5,9.5,10.5) | (180,205,230) | 0.655 | 0.848 | 0.725 | 1 | 0.403 |
| Opt. 3.2 | (1/500−1/2000, 1/500, 1/500+1/2000) | $f_{500}$ | $\infty$ | (9,10,11) | (200,225,250) | 0.829 | 0.853 | 1 | 0.7 | 0.495 |
| Opt. 3.3 | (1/850−1/2000, 1/850, 1/850+1/2000) | $f_{850}$ | $\infty$ | (11,12,13) | (275,300,325) | 0 | 0.678 | 1 | 0 | 0 |

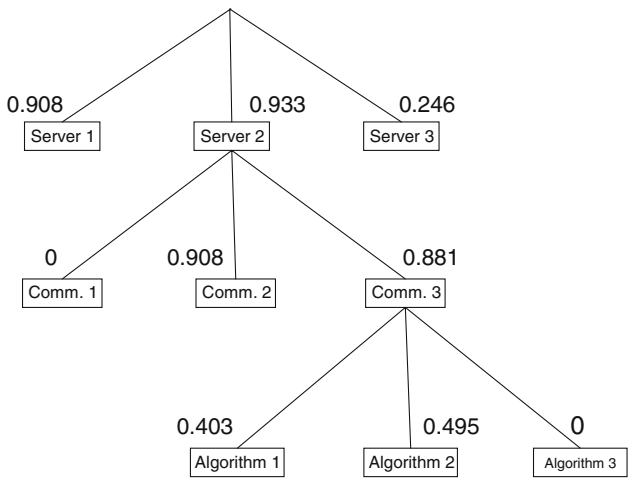



**Fig. 8** Design tree with imperfect estimations and requirements

which is much closer to the intuition, since the estimations for both alternatives were also very close.

## 6 Tool support

Tracing design decisions and evaluating design alternatives with imperfection models is very labor intensive and therefore automatic support is necessary. To assist the software engineer in its application, we have implemented our approach in a tool prototype. The architecture of our tool is shown in Fig. 9. Here, the models and processes are represented as rectangles and ellipses, respectively.

In the Design Decision Tracer the stakeholder provides the initial quality requirements specification for the system

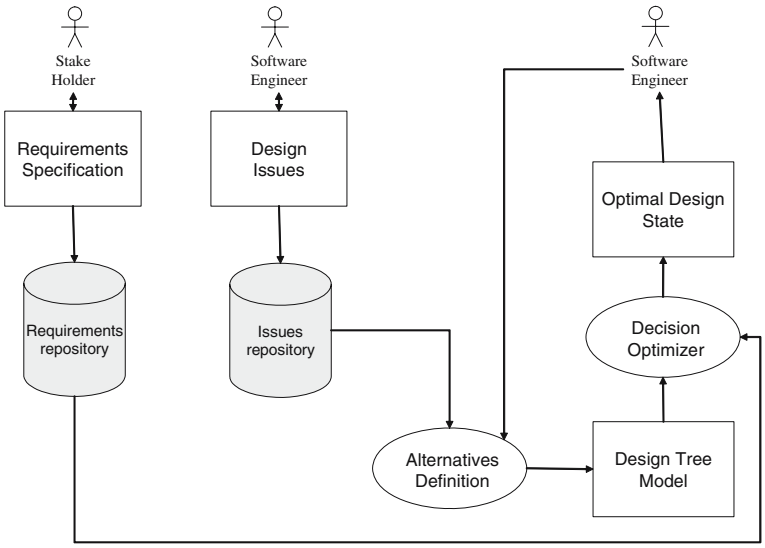



**Fig. 9** DecisionTracer tool architecture

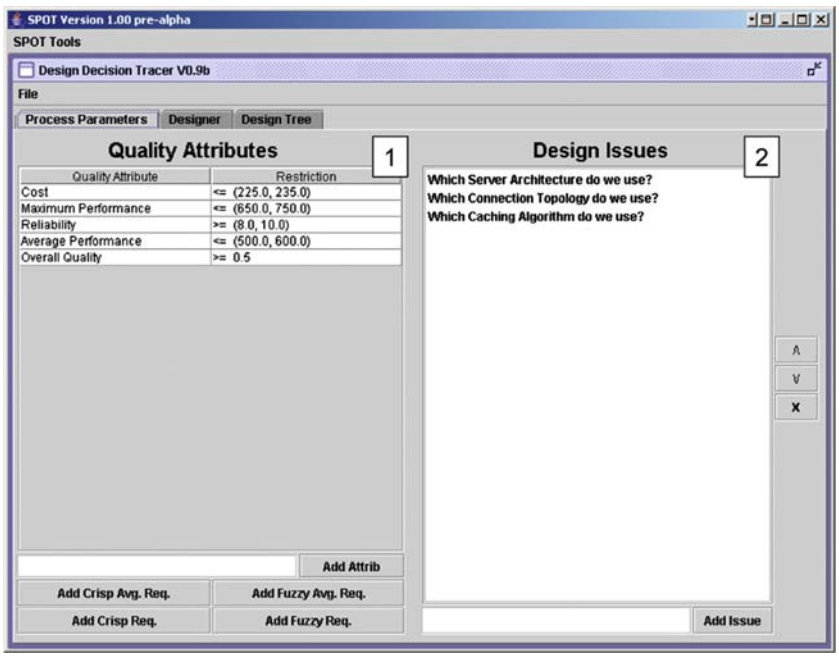



**Fig. 10** Process parameters tab

that should be designed. The software engineer identifies the design issues that should be resolved. The second step is the identification of design alternatives for each individual design issue. The Alternatives Definition process, with the help of the software engineer, identifies the alternatives for the current design issue, and estimates their respective quality attributes. After all the alternatives for the current design issue have been defined, the Design Tree Model is updated to reflect this new state of knowledge. The Decision Optimizer determines the best design state from which to continue the design process. The Optimal Design State result is presented to the software engineer, who now can continue with the next design issue.

The DecisionTracer is comprised of three user tabs, the Process Parameters Tab, the Designer Tab and the Design Tree Tab.

The first tab in the DecisionTracer is the Process Parameters tab, depicted in Fig. 10. In this tab the general parameters for the design process are defined, being the quality requirements that should be fulfilled and the design issue that should be resolved. The tab is divided into two parts, the Quality Attributes part (1), and the Design Issues part (2).

The Designer Tab in Fig. 11 is the tab at which the design issues are resolved in sequence. At 1 the current design issue is displayed, and below it is the list of design issues that should be resolved after the current one is completed. It is

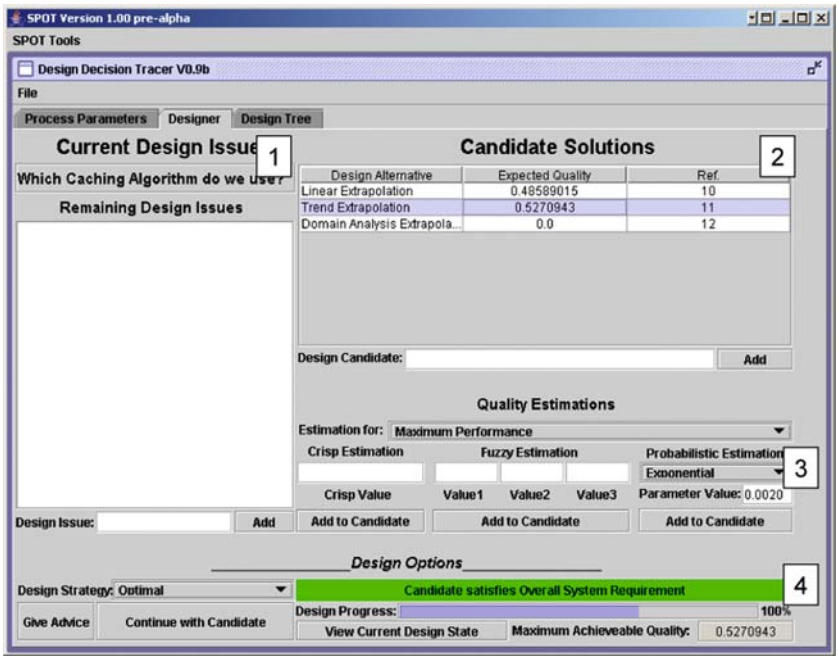



**Fig. 11** Designer tab

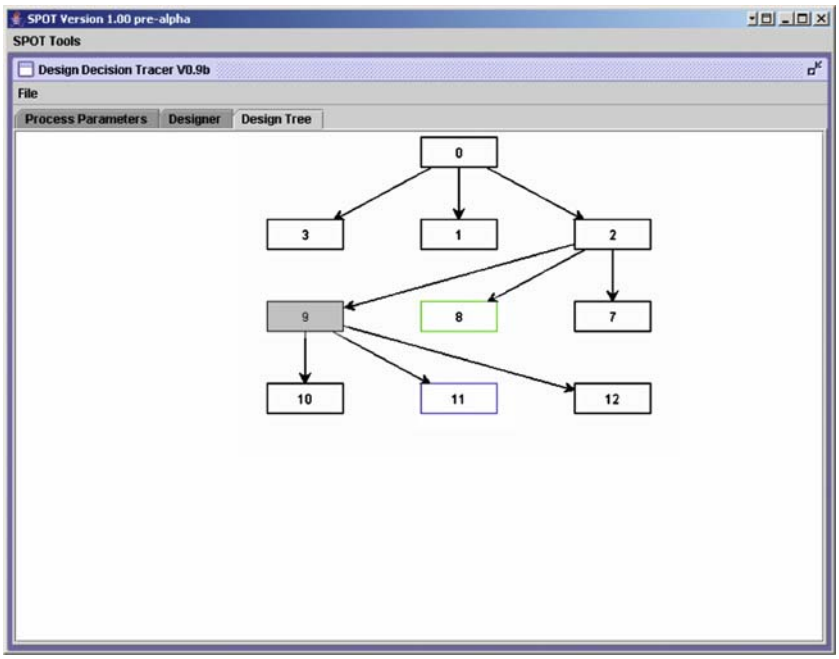



**Fig. 12** Design tree tab

possible to enter candidate solutions for the current Design Issue at 2. For each candidate it is possible to enter the estimated quality of each quality attribute using fuzzy or probabilistic models at 3. Using the controls at 4 the design state can be examined and it is possible to ask the tool to offer decision support.

In the final tab the software designer find the design tree, depicted in Fig. 12. While it is not necessary to understand the Design Tree approach to use the Decision-Tracer, it is possible to inspect the design tree of the current situation at any time in the design tree tab. In the design tree one node is colored grey (9). This is the node that represents the current design state. In addition there is a green node (8), which is the best node to continue from according to the Optimal Design Strategy. The blue node (11) is the best node according to the Smart Design Strategy.

To analyze the scalability of our approach we are conducting experiments where the toolset is applied in an industrial setting. In particular the added workload and the benefits of our approach are analyzed in these experiments. The first pre-

liminary results indicate an increased insight that is gained from explicitly modeling design decisions and alternatives. Additionally, after a short introduction the possibilities of specifying imperfect requirements and imperfect estimations became quite natural to the users. The toolset is a valuable addition when evaluating design alternatives.

## 7 Related work

### 7.1 Decision models of software processes

During the last 20 years, a considerable number of design methods have been introduced, such as Structural design (Yourdon and Constantine 1979) and Rational Unified Process (Jacobson et al. 1999). These approaches generally differ from each other with respect to the adopted models (functional, data-oriented, object-oriented, etc.). These methods propose a process which is guided by a large set of explicit and implicit heuristics rules. A method may distinguish itself from the others by introducing and emphasizing its own design heuristics. In Tekinerdogan and Akşit (2002), based on their heuristics, architecture design methods are classified as *artifact-driven*, *use-case driven* and *domain-driven*. In the artifact-driven approaches, software is designed from the perspective of the available software artifacts. For example, in the OMT method, a class is identified using the rule: *"If an entity in the requirement specification is relevant then select it as a tentative class"*. In the use-case driven approaches, use cases are applied as the primary artifacts in designing software systems. For example, in RUP, analysis packages, which are the primary means to decompose software, are identified with the rule: *"Identify the analysis packages if use cases are required to support a specific business process"*. In the domain-driven approaches, the fundamental software components are extracted from the concepts of the domain model.

An extensive number of software engineering environments have been proposed to support software engineering methods. Most environments provide model editing, consistency checking, version management and code generation facilities. There is a considerable amount of research on process modeling (Kaiser et al. 1994; Finkelstein et al. 1994), as well as research in the field of assisting software designers with automated reasoning mechanisms. However, formalizing design heuristics and providing some sort of expert system support during the design process is not exploited well. This is in particular because most approaches can not deal with imperfect information in the design process. In Akşit and Marcelloni (2001b), a design heuristics support approach based on fuzzy logic is proposed. However, this work does not address the same problem of imperfect information as defined in this paper.

### 7.2 Modeling imperfect information in design processes

Modeling imperfection in the inputs of design processes is not new, however it is seldomly applied in the field of software design. The most well-known area in software engineering in which the potential consequences of imperfect information are considered is risk management (Karolak 1995). In this area the influence of probabilistic events is analyzed in for instance software design processes. However, the techniques that are proposed in this field address a different type of imperfection than our approach. In our approach we try to facilitate imperfection in requirement specifications and quality estimations, and we have identified different types of imperfection that can occur. As such, our approach is not in particular a risk management approach, but rather a refinement of software development activities. In Akşit and Marcelloni (2001b) fuzzy logic is applied to support the partial applicability of design heuristics in the OMT development process. By applying fuzzy reasoning techniques, the inconsistency can be controlled and maintained to a point where it can be resolved by new design input. In Yen and Lee (1993) a fuzzy logic framework is defined that can be used to model imprecise functional requirements. After each design step the proposed solution can be compared with the requirement, similar to proving an invariant over a piece of code. The resulting value then indicates to which degree the requirement holds.

In Liu and Da (2005) an extension to decision trees (see next paragraph) is proposed. The (imprecise) attitude of the decision maker with respect to risks is modeled using techniques from fuzzy logic, and combined with the decision optimization algorithms of probabilistic decision trees. In Law and Antonsson (1995), an approach is proposed to model imprecision in design inputs. This imprecision is captured using fuzzy set theory, and is then used to explore the possible design alternatives based on this model. In addition, the method defines means to evaluate design alternatives based on these models. In Noppen et al. (2004), the uncertainty of market demands for software products is captured using probabilistic models. These models are then used by a Markov decision model to determine the implementation order of the components of the system, in order to optimize the expected profit.

### 7.3 Traceability of design decisions in software engineering

Keeping track of the design decisions that are taken during the design process is not new. In the field of requirements traceability the relationship between intermediate design artifacts and the originating requirement(s) are made explicit. The models that have been proposed in this field can be classified according to the specific type of information they aim to capture, such as functional or non-functional tracing, forward or backward tracing, etc. In the case the design trees are used to

capture design decisions, the solution is in the area of non-functional requirements tracing.

Most approaches that have been proposed to trace design decisions are based on decision tree models. In Potts and Bruns (1988) and Ran and Kuusela (1996) alternative approaches are described for capturing design decisions and their motivations, which are similar to the design tree approach. Design artifacts are captured using a graph structure, as well as relevant design considerations. However, contrary to the design tree approach, it is not possible to traverse the graph structure in order to decide on the subsequent design trajectory in a configurable manner, as is possible with the design tree approach. In Cimitile et al. (1992) design decisions are recorded as annotations to enhance software maintenance. These annotations are used to trace the decisions that have been taken with respect to transforming an intermediate design into a design for a specific implementation language. In general, it can be said that requirement traceability approaches are custom made for a particular application area, and contain domain-specific model attributes. This makes it difficult to reuse and compare traceability models. For this purpose work has been done in creating reference models for requirements traceability (Ramesh and Jarke 2001). The model aims to provide the relevant elements of a traceability model, to which only the domain specific elements need to be added. By adjusting the design tree model to conform to the basic elements of the reference model, these respective qualities can also be achieved.

In the area of design rationale management, many approaches have been proposed, which can be used to capture relevant information that is the result of the design process. Depending on the nature of the design process, different types of approaches are used. In Regli et al. (2000) a distinction is made between feature-oriented and process-oriented approach for capturing design rationale. In feature-oriented approaches, design rationale is captured in domains that are well-known and standardized. Process-oriented approaches emphasize design rationale as a history of the design process, and are used in domains where problems are vague and the solutions poorly understood. Examples of such approaches are Lee and Lai (1991) and McCall (1991). However, where design rationale management approaches aim to capture the intuitions and validations of design decisions, they do not explicitly consider imperfection, even when they are aware of the fact that the domain contains this. In the Design Tree model the design rationale based on the expected quality for each design decision is captured, and the approach supports imperfection in both requirements and estimations.

Overall, while all these models capture relationships between artifacts that are the results of design activities, and while a number of the models mentioned overlap with the optimization approach the design tree approach offers, they do not explicitly consider the possibility of imperfect information. In the design tree approach imperfection in both quality requirements and quality estimations is allowed, without compromising the capabilities of the optimization approach.

## 8 Discussion and conclusions

### 8.1 Discussion

In Sect. 2, imperfect information in software development and making the right design decision were identified as two important problems. We have presented a method that can help designers in evaluating design alternatives and optimize the selection of design alternatives using imperfect information. In the following we evaluate our approach with respect to a number of concerns:

- *Can imperfect design inputs be avoided with more complex design methods?*
  An alternative solution to the proposed approach can be to improve the models that are currently used, such that the imperfection will not manifest itself in the design inputs, such as requirement specifications and quality estimations. This could be done, for instance, by including domain information for the interpretation of requirements and estimations. However, although this can address the problems in specific cases, the imperfection in the shape of imprecision and/or uncertainty that can occur during the design process is unavoidable. When specifying requirements, due to external circumstances it is sometimes impossible to provide precise descriptions. In the example case, for instance, it is argued that for budgeting a certain tolerance range exists, which means the precise budget restrictions are just not known. Additionally, quality estimations are intrinsically uncertain, since the necessary information for performing measurements is just not available. This actually was the reason to make estimations rather than measurements. This uncertainty can not be prevented by additional deliberation. The failure of the waterfall-model, and the general acceptance of iterative and agile approaches is a clear illustration of this fact.
- *Can imperfect design inputs be avoided by more experienced designers?*
  Another approach to address these problems could be to improve the skills of the software architect. By training the architect in recognizing imperfection in design input, the imperfection can be removed and/or resolved before the design process is continued. By facilitating the negotiations with the stakeholders imperfect information can be reduced to a minimum. However, as was already mentioned above, imperfection in design input is not some-

thing that can always be avoided. The information given by the stakeholders can depend on external events, which makes it difficult to remove imperfection. For instance, in the example case the exact budget might not yet have been established at the start of the project. Therefore the budgetary restrictions are prone to imperfect specifications that can not be removed at that point, even when the imperfection is identified. This is even more the case for quality estimations that should be made of software systems that have not been implemented. While using more experienced designers will increase the accuracy of these estimations, they will still be prone to imperfection due to the lack of information. Nonetheless it will be very useful for architects to identify imperfect information when it occurs, since this will make it possible to treat the imperfection more appropriately.

- *How do we acquire the applicable fuzzy sets and probability distributions?*
  The accuracy of the proposed approach depends heavily on whether the fuzzy set models and probability distributions capture the nature of the imperfection appropriately. This in turn means that it is very important to define the "right" fuzzy sets and probability distributions to represent the imperfection. How can these fuzzy sets and probability distributions be identified? While it is true that the use of imperfection models adds an extra level of insight, in our early experiments the definitions were quite natural for the users. The variance that might exist in quality requirements such as performance was easily captured by defining the boundaries of triangular fuzzy numbers. In addition, it should be noted that probability distributions have long been used to model for instance performance of computer systems with a probabilistic nature. Additionally, fuzzy set theory offers the possibility to use linguistic variables (Zadeh 1975) to refer to standard definitions of fuzzy sets within a particular area. This can be used to model domain specific information. In this way generic information can be captured and processed by abstracting away from the mathematical definition of the fuzzy sets. Also the support of tools can help greatly, since it facilitates experimentation with specific fuzzy sets and probability distributions. While the actual definition of the fuzzy sets and/or probability distributions is by no means trivial, it is important to note that in current methods all kinds of crisp design rules are used which are not justifiable. For example, in requirements specification methods nouns are identified as candidate classes, even while this is not generally accurate. From this perspective, using probability theory and fuzzy set theory, while requiring additional insights, can be considered more precise.
- *Can fuzzy logic/fuzzy set theory and probability theory model the imperfection appropriately?*
  In our approach we use models from probability theory and fuzzy set theory to model the imperfection that can occur in the software development process. It can be questioned how well these models are able to capture the nature of the imperfection that can occur in design information. The nature of the imperfection does not necessarily correspond to the way in which imperfection is modeled in probability and fuzzy set theory. But while it is true that these models do not always reflect the actual nature of imperfection that can be found, it is certain that these model address the issue of imperfection more accurately than ignoring the imperfection in the design information, and trying to resolve it at later stages solely by iteration and incremental design. While probability and fuzzy set theory can only cover part of the imperfection that can occur in design information, at least it can be assured that for this part the imperfection is treated appropriately. In other words probability theory and fuzzy set theory is the best we have, and is by far better than ignoring imperfection altogether.
- *Will the proposed approach scale up in industry-sized design processes?*
  Finally the application of our approach in practice creates extra overhead for the software engineer. Keeping track of all design decisions that have been made using the design tree increases the workload. Similarly, the explicit evaluation of design alternatives using imprecise requirements and uncertain evaluations forces the architect to perform additional activities, which can become computationally cumbersome. However, most of these activities are extensions of activities that are performed during the software design process, either explicitly or implicitly. For instance, in normal design processes design alternatives are compared and evaluated, even while this is not made explicit. By supporting the capturing of the design decisions in an automated tool, the additional effort is minimized. In this tool also the computational support for comparing different types of imperfect information is included, which minimizes the computational overhead for the software engineer. This way, the extra activities required for the application of this approach do not create overhead that is larger than the gains. To analyze our approach we are applying our approach and toolset to an industrial case study. The initial results indicate the usefulness and scalability of our approach.

## 8.2 Conclusions

In Sect. 2 imperfect information in quality requirements and quality estimations and the cascading of errors in design decisions were identified as two important problems in the design of software systems. The first problem can lead to making

wrong decisions during the design process, since quality assessments do not represent the current situation accurately. The second problem is a direct consequence of the sequential nature in which design decisions are taken, which causes errors in individual decisions to influence the correctness of all decisions hereafter. Additionally, even when it becomes clear that the current design has not been the right choice, it is not easy to step back through the design process and to determine the point from which to continue.

We have shown that imperfect information can be managed by capturing the nature of the imperfection. To accomplish this, we have made the explicit distinction between impreciseness in quality requirements and uncertainty in quality estimations. By capturing both types of imperfect information and their specific character with applicable models such as probability theory or fuzzy set theory, the design alternatives considered during the design process can be evaluated more accurately. Furthermore, the means of comparing different types of imperfect information are defined, to enable the software engineer to evaluate design alternatives in much the same manner as in current approaches. This has been demonstrated by applying the approach to an example case, where two design alternatives were estimated to have very similar qualities. In the traditional evaluation method one alternative was evaluated as being unsatisfactory, since the quality attributes were just outside the quality constraints. When it was evaluated using our approach, the alternatives showed that the quality was quite comparable, much like what should be expected.

In addition we have shown that the design process can be supported by tracing the design decisions that are made. For each design decision the considered alternatives are logged, and the evaluations are made explicit. To accomplish this, a distinction is made between the quality requirements, which restrict the allowed behavior of the system, and the quality estimations, which describe the expected behavior of the system. The relationship between these elements is captured by a tree structure, which can be traversed in an algorithmic manner so that the design space can be explored systematically.

The approach that has been presented in this paper requires a considerable amount of computations to be performed. To support the software engineer in the application of this approach, tooling is planned. In addition domain knowledge that contains impreciseness can be modeled and supported by the tooling. This can for instance be done for well-known domains such as real-time computation. This can minimize the impreciseness modeling and evaluation the choice of the most appropriate parameters.

**Acknowledgements** We would like to thank the anonymous reviewer for the valuable feedback on the initial version of this article.

## Appendix A: Probabilistic and fuzzy techniques

The means to model imperfect information that are used are probability theory, fuzzy set theory and fuzzy probability theory. In this appendix a description is given of the basic concepts that are used.

### A.1 Probability density functions

A common approach to model impreciseness is the use of probability density functions. Given a probability density function $f$, the probability $P(a, b)$ of the occurrence of an event between $a$ and $b$ is given by

$$P(a,b) = \int_a^b f(x)\mathrm{d}x$$

and the expectation value $E$ of the occurrence is given by

$$E = \int_U xf(x)\mathrm{d}x$$

where $U$ is the universe of discourse.

For the example exponential density functions are used, given by $f_\lambda(x) = \lambda * e^{-\lambda x}$ where $\lambda > 0$.

For this choice the probability $P_\lambda(a, b)$ of the occurrence of an event between $a$ and $b$ is given by

$$P_\lambda(a,b) = \int_a^b f_\lambda(x)\mathrm{d}x = \mathrm{e}^{-\lambda b} - \mathrm{e}^{-\lambda a}$$

and the expectation value $E_\lambda$ is given by

$$E_\lambda = \int_0^\infty xf_\lambda(x)\mathrm{d}x = 1/\lambda$$

### A.2 Fuzzy numbers

Fuzzy sets and fuzzy numbers have been introduced in Sect. 3.2. For $0 < \alpha \leq 1$, the $\alpha$-cut of a fuzzy set F with membership function $\mu$ is defined by $\{\mathrm{x}|\mu(\mathrm{x}) \geq \alpha\}$ and denoted by $F[\alpha]$. If F is a fuzzy number, then $F[\alpha]$ is an interval.

The operation on fuzzy sets which is needed in this paper is the comparison of a fuzzy estimation $C$ with a fuzzy requirement $A$. The result of the comparison should be a number between zero and one, indicating the degree to which $A$ is smaller than $C$.

This degree is defined to be $\int_0^1 S(\alpha)\mathrm{d}\alpha$, where $S(\alpha)$ is the degree to which $A[\alpha]$ is smaller than $C[\alpha] \cdot S(\alpha)$ is defined to be the fraction of $C[\alpha]$ which belongs to $A[\alpha]$. With $C[\alpha] =$

$[q(\alpha), r(\alpha)]$ and $A[\alpha] = ]-\infty, p(\alpha)]$ we find

$$S[\alpha] = 0 \quad \text{if } r(\alpha) \leq p(\alpha)$$
$$S[\alpha] = 1 \quad \text{if } p(\alpha) \leq q(\alpha)$$
$$S[\alpha] = (p(\alpha) - q(\alpha))/(r(\alpha) - q(\alpha)) \quad \text{if } q(\alpha) < p(\alpha) < r(\alpha)$$

This will be elaborated further in Appendix B.6, for the case the estimations are expressed using triangular fuzzy numbers.

### A.3 Fuzzy probability theory

In the previous two paragraphs two different types of impreciseness models have been described. Each of these models is aimed at specific types of impreciseness. However, this does not exclude one type of impreciseness occurring in the application of the other model. Fuzzy Probability Theory combines impreciseness modeling using probability theory with the extra possibility to capture impreciseness with respect to the actual parameters of the probability distribution.

Recently there has been an increased interest in the fuzzy logic community in the area of *fuzzy probabilities*. In Buckley (2003) fuzzy probability distributions are defined, by replacing parameters in families of crisp probability distributions (such as exponential, standard normal, etc.) with fuzzy values.

Consider, like before, the family of exponential probability density functions, given by $f_\lambda(x) = \lambda * e^{-\lambda x}$ where $\lambda > 0$. For a fuzzy number $\Lambda$ on the domain of non-negative real numbers, the fuzzy exponential probability distribution $f_\Lambda$ gives fuzzy results, whose $\alpha$-cuts are obtained from the $\alpha$-cuts of $\Lambda$. The probability $P_\Lambda(a, b)$ of the occurrence of an event between $a$ and $b$ is given by

$$P_\Lambda(a, b)[\alpha] = \left\{ \int_a^b f_\lambda(x) | \lambda \in \Lambda[\alpha] \right\}$$

and the expectation value $E_\lambda$ is given by

$$E_\Lambda[\alpha] = \left\{ \int_0^\infty x f_\lambda(x) | \lambda \in \Lambda[\alpha] \right\}$$

## Appendix B: Comparing imperfect estimations with imperfect requirements

In software design processes, it is common to compare estimations of the expected quality for the identified design alternatives with the quality requirements. In the case where both the requirements and the estimations are expressed in a crisp way, this can be achieved in a straightforward manner. However, since for instance fuzzy requirements and probabilistic estimations express impreciseness with different models, the operations for comparison need to be defined. In the remainder of this appendix, for each combination of impreciseness types these operations will be defined. Additionally, a distinction is made between two types of restrictions, a restriction on the allowed *average* indicated by $a$, and a restriction on the allowed *maximum* indicated by $b$. The estimation is indicated with a $c$. We will define two functions for comparing requirements to estimations: $\text{Comp}_{\text{avg}}$ for comparing estimations to restrictions on the average and $\text{Comp}_{\text{max}}$ for comparing estimations to restrictions on the maximum. The functions take a requirement and an estimation as parameters, and return a number between 0 and 1, which represents the degree to which the estimation is smaller than the requirement. Since $\text{Comp}_{\text{max}}$ only differs from $\text{Comp}_{\text{avg}}$ in case of probabilistic or fuzzy probabilistic estimations, for crisp and/or fuzzy estimations both functions will be denoted by Comp.

### B.1 Comparing crisp requirements with probabilistic estimations

Suppose the estimation is a probability density function $f$. Then

$$\text{Comp}_{\text{avg}}(a, f) = \begin{cases} 1 & \text{if } \int_U x f(x) \mathrm{d}x \leq a \\ 0 & \text{otherwise} \end{cases}$$

$\text{Comp}_{\text{max}}$ is defined to be the probability that the requirement is fulfilled:

$$\text{Comp}_{\text{max}}(b, f) = \int_{x \leq b} f(x) \mathrm{d}x$$

### B.2 Comparing crisp requirements with fuzzy estimations

Let the fuzzy estimation be the triangular fuzzy number $(c_1, c_2, c_3)$ and the crisp requirement be that the number is smaller than the number a. The value of $\text{Comp}(a, (c_1, c_2, c_3))$ is given in Table 13.

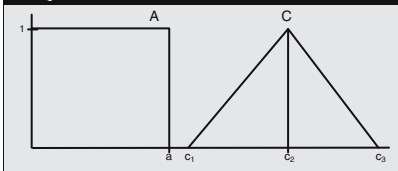
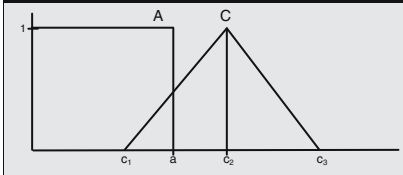
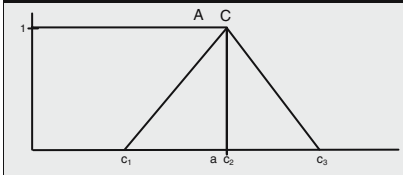
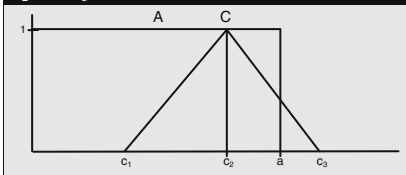
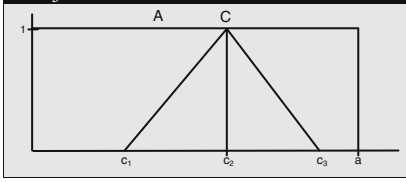
The expressions in the table are special cases of the more general expressions that compare fuzzy requirements with fuzzy estimations. These expressions and their derivation can be found in sect. B.6.

### B.3 Comparing crisp requirements with fuzzy probabilistic estimations

Suppose the estimation is a fuzzy probability distribution with density function $f_P$. Its expectation value is a fuzzy number $E$, which $\alpha$-cuts are given by

$$E_p[\alpha] = \left\{ \int_0^\infty x f_p(x) | p \in P[\alpha] \right\}$$

**Table 13** Comparing crisp requirements with fuzzy estimations [FX]

| | |
|---|---|
| $a \leq c_1$ | |
|  | 0 |
| $c_1 < a \leq c_2$ | |
|  | $\frac{a - c_1}{c_3 - c_1} - \frac{c_2 - a}{c_3 - c_1} \ln\left(\frac{a - c_1}{c_2 - a} + 1\right)$ |
| $c_2 = a \ \& \ a \leq c_3$ | |
|  | $\frac{a - c_1}{c_3 - c_1}$ |
| $c_2 < a \leq c_3$ | |
|  | $\frac{a - c_1}{c_3 - c_1} + \frac{a - c_2}{c_3 - c_1} \ln\left(1 + \frac{c_3 - a}{a - c_2}\right)$ |
| $a > c_3$ | |
|  | 1 |

$\text{Comp}_{\text{avg}}$ can therefore be defined similarly to the function for comparing crisp requirements and fuzzy estimations. $\text{Comp}_{\text{avg}}$ can now be worked out according to the definitions in Appendix A.2.

$\text{Comp}_{\max}(b, f_P)$ is defined to be the defuzzification of the fuzzy number $Q$, which $\alpha$-cuts are given by

$$Q[\alpha] = \left\{ \int_{x \leq b} f_p(x)\mathrm{d}x \,|\, p \in P[\alpha] \right\}$$

## B.4 Comparing fuzzy requirements with crisp estimations

Comparing fuzzy requirements with crisp estimations can be done by noting that a crisp number c is the same as the fuzzy triangular number $(c, c, c)$. Suppose the requirement is semi-trapezoidal number $(a_1, a_2)$.

$$\text{Comp}((a_1, a_2), c) = 0 \quad \text{ifc} \ \leq \text{a}_1$$
$$\text{Comp}((a_1, a_2), c) = \frac{a_2 - c}{a_2 - a_1}, \quad \text{if} \ a_1 < c \leq a_2$$
$$\text{Comp}((a_1, a_2), c) = 1 \quad \text{ifc} \ > a_2$$

This function is a special case of the more general function that is used to compare fuzzy requirements with fuzzy estimations, given in Sect. B.7.

## B.5 Comparing fuzzy requirements with probabilistic estimations

Suppose the estimation $c$ is given by density function $f$ with expectation value $m$.

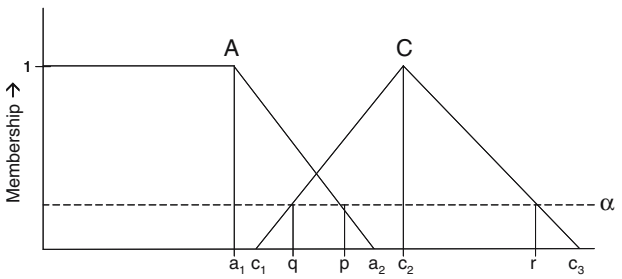
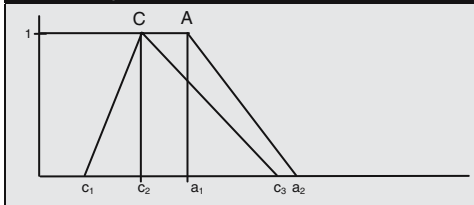
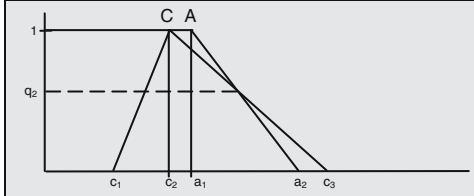
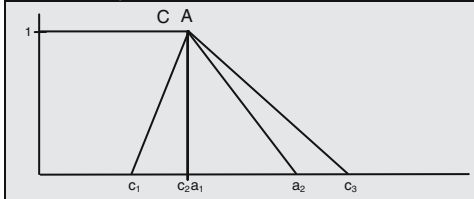
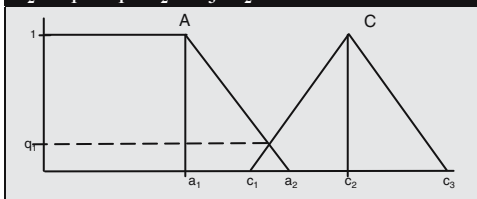
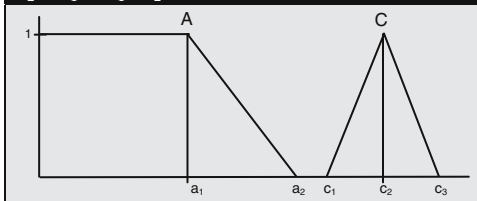
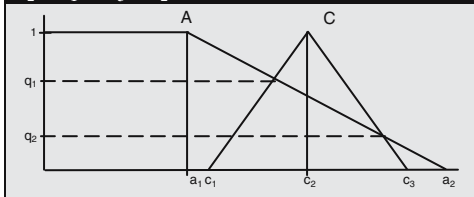



**Fig. 13** Fuzzy numbers

**Table 14** Comparing fuzzy requirements with fuzzy estimations [FX]

| | |
|---|---|
| **$c_2 \le a_1$ & $c_3 \le a_2$** | |
|   | $S(\alpha) = 1$ , for $0 \le \alpha \le 1$ |
| | $Comp((c_1, c_2, c_3), (a_1, a_2)) = 1$ |
| **$c_2 < a_1$ & $c_3 > a_2$** | |
|   | $\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$ , for $0 \le \alpha \le q_2$ |
| | $1$ , for $q_2 \le \alpha \le 1$ |
| | $\frac{a_2 - c_1}{c_3 - c_1} + \frac{a_1 - c_2}{c_3 - c_1} \ln\left(1 + \frac{c_3 - a_2}{a_1 - c_2}\right)$ |
| **$c_2 = a_1$ & $c_3 > a_2$** | |
|   | $\frac{a_2 - c_1}{c_3 - c_1}$ , for $0 \le \alpha \le 1$ |
| | $\frac{a_2 - c_1}{c_3 - c_1}$ |
| **$c_2 > a_1$ & $c_1 < a_2$ & $c_3 \ge a_2$** | |
|   | $\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$ , for $0 \le \alpha \le q_1$ |
| | $0$ , for $q_1 \le \alpha \le 1$ |
| | $\frac{a_2 - c_1}{c_3 - c_1} - \frac{c_2 - a_1}{c_3 - c_1} \ln\left(\frac{a_2 - c_1}{c_2 - a_1} + 1\right)$ |
| **$c_2 > a_1$ & $c_1 \ge a_2$** | |
|   | $0$ , for $0 \le \alpha \le 1$ |
| | $0$ |
| **$c_2 > a_1$ & $c_3 < a_2$** | |
|   | $1$ , for $0 \le \alpha \le q_2$ |
| | $\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$ , for $q_2 \le \alpha \le q_1$ |
| | $0$ , for $q_1 \le \alpha \le 1$ |
| | $1 + \frac{c_2 - a_1}{c_3 - c_1} \ln\left(1 - \frac{c_3 - c_1}{c_2 - a_1 + a_2 - c_1}\right)$ |

$Comp_{avg}$ is equal to the comparison of a fuzzy requirement $a$ to a crisp estimation $m$:

When comparing crisp requirements to probabilistic estimations the following expression is used: $\int_{x \le b} f(x)\mathrm{d}x$. This can be written as $\int_0^\infty f(x)\mu(x)\mathrm{d}x$, with $\mu(x) = 1$ for $0 \le x \le b$ and $\mu(x) = 0$ otherwise. This means that $\mu(x)$ is the membership function of the set of allowed values. When this is extended to fuzzy requirements, and we have a fuzzy maximum requirement with membership function $B$ we get

$$\mathrm{Comp}_{\max}(B, f) = \int_0^\infty B(x) f(x) \mathrm{d}x$$

B.6 Comparing fuzzy requirements with fuzzy estimations

In Fig. 13 two fuzzy numbers are depicted, which should be compared. The degree to which a fuzzy estimation $C$ is smaller than a fuzzy requirement $A$ is defined to be $\int_0^1 S(\alpha)\mathrm{d}\alpha$, where $S(\alpha)$ is the degree to which $A[\alpha]$ is smaller than $C[\alpha]$. $S(\alpha)$ is defined to be the fraction of $C[\alpha]$ which belongs to $A[\alpha]$. With $C[\alpha] = [q(\alpha), r(\alpha)]$ and $A[\alpha] = ]-\infty, p(\alpha)]$ we find:

$$S(\alpha) = 0 \quad \text{if } r(\alpha) \leq p(\alpha)$$
$$S(\alpha) = 1 \quad \text{if } p(\alpha) \leq q(\alpha)$$
$$S(\alpha) = (p(\alpha) - q(\alpha))/(r(\alpha) - q(\alpha)) \quad \text{if } q(\alpha) < p(\alpha) < r(\alpha)$$

In this article fuzzy estimations are assumed to be triangular fuzzy numbers $(c_1, c_2, c_3)$, and requirements are assumed to be semi-trapezoidal fuzzy numbers $(-\infty, a_1, a_2)$, as can be seen in Fig. 9. In this case, $p(\alpha)$, $q(\alpha)$ and $r(\alpha)$ are given by

$$p(\alpha) = a_2 - \alpha(a_2 - a_1)$$
$$q(\alpha) = c_1 + \alpha(c_2 - c_1)$$
$$r(\alpha) = c_3 - \alpha(c_3 - c_2)$$

Let the fuzzy estimation be the triangular fuzzy number $(c_1, c_2, c_3)$ and the fuzzy requirement be that the number is smaller than the semi-trapezoidal fuzzy number $(a_1, a_2)$. We define $\mathrm{q}_1$ and $\mathrm{q}_2$ as follows:

$$q_1 = \frac{a_2 - c_1}{a_2 - a_1 + c_2 - c_1} \quad q_2 = \frac{a_2 - c_3}{a_2 - a_1 - c_3 + c_2}$$

Then $q_1$ is the $\alpha$-height of the intersection of the lefthand side of the estimation with the requirement, and $q_2$ is the $\alpha$-height of the intersection of the righthand side of the estimation with the requirement.

$S(\alpha)$ is defined for six cases of comparing a requirement $(a_1, a_2)$ with an estimation $(c_1, c_2, c_3)$. The value of Comp( $(a_1, a_2)$, $(c_1, c_2, c_3)$) is given in Table 14. In the table for each of the six cases a graphical depiction of the actual situation is given in the left column. In the right column, for each situation $S$ is defined for height $\alpha$ above the line, and $\mathrm{Comp}((a_1, a_2), (c_1, c_2, c_3)) = \int_0^1 S(\alpha) d\alpha$ is given below the line.

B.7 Comparing fuzzy requirements with fuzzy probabilistic estimations

Suppose the estimation c is given by density function $f_P$ with average $E$, *given by*

$$E_P[\alpha] = \left\{ \int_0^\infty x f_p(x) | p \in P[\alpha] \right\}$$

The expectation value of the fuzzy probability distribution is a fuzzy number $E$, therefore $\mathrm{Comp}_{\mathrm{avg}} = \mathrm{Comp}(\mathrm{A}, E_P)$ as defined in Sect. B.6. Note that the expressions in B.6 hold for the case that $\mathrm{E}_P$ is a triangular number. As this will in general no be the case, either the procedure given in Appendix A.2 should be used, or $E_P$ must be approximated by a fuzzy triangular number.

The outcome of a value in a fuzzy probability distribution is a fuzzy degree. This means a computation similar to the computation with crisp probability distributions can be performed for maximum requirement $b$.

For fuzzy requirements and crisp probabilistic estimations the resulting function $\mathrm{Comp}_{\max}$ is defined as

$$\mathrm{Comp}_{\max}(B, f) = \int_0^\infty B(x) f(x) \mathrm{d}x$$

Similarly, $\mathrm{Comp}_{\max}(B, f_P)$ is defined to be the defuzzification of the fuzzy number $Q$, which $\alpha$-cuts are given by

$$Q[\alpha] = \left\{ \int_0^\infty B(x) f_p(x) \mathrm{d}x | p \in P[\alpha] \right\}$$

## References


Akşit M, Marcelloni F (2001) Deferring elimination of design alternatives in object-oriented methods. In: Concurrency and computation: practice and experience, Wiley, Chichester, pp 1247–1279

Akşit M, Marcelloni F (2001) Leaving inconsistency using fuzzy logic, information and software technology 43(10):725–741

Buckley JJ (2003) Fuzzy probabilities new approach and applications. Springer, Heidelberg

Cimitile A, Lanubile F, Visaggio G (1992) Traceability based on design decisions. In: Proceedings of conference on software maintenance. IEEE Press, New York, pp 309–317

Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J (2004) Documenting software architectures. Addison Wesley, Reading

Finkelstein A, Kramer J, Nuseibeh B (1994) Software process modelling and technology. Research Studies Press Ltd

Jacobson I, Booch G, Rumbough J (1999) The unified software development process. Addison Wesley, Reading

Kaiser GE, Popovich S, Ben-Shaul IZ (1994) A bi-level language for software process modeling. In: Tichy W (eds) Configuration management. Wiley. Chichester, pp 39–72

Karolak DW (1995) Software engineering risk management. Wiley, Chichester. ISBN: 978-0-8186-7194-4

Kazman R, Klein M, Barbacci M, Longstaff T, Lipson H, Carriere J (1998) The architecture tradeoff analysis method. In: 4th International conference on engineering of complex computer systems. IEEE Computer Society Press, Los Alamitos, pp 68–78

Law WS, Antonsson EK (1995) Optimization methods for calculating design imprecision. In: Advances in design automation. ASME, New York. pp 471–476

Lee J, Kuo J, Hsueh N, Fanjiang Y (2003) Trade-off requirement engineering. In: Lee J. (eds) Software engineering with computational intelligence. Springer, Heidelberg, pp 51–72

Lee J, Lai K (1991) What's in design rationale. Human–Computer Interaction 6(3–4):251–280

Lethbridge TC, Laganière R (2005) Object-oriented software engineering practical software development using UML and Java. McGraw Hill, New York

Liu X, Da Q (2005) A Decision tree solution considering the decision maker's attitude. In: Fuzzy sets and systems. Elsevier, Amsterdam, pp 437–454

McCall RJ (1991) PHI: A conceptual foundation for design hypermedia. Design Stud 12(1):30–41

Noppen J, Akşit M, Nicola V, Tekinerdogan B (2004) Market-driven approach based on markov decision theory for optimal use of resources in software development. IEE Proc Softw 151(2): 85–94

Potts C, Bruns G (1988) Recording the reasons for design decisions. In: ICSE '88: Proceedings of the 10th international conference on software engineering. IEEE Computer Society Press, Los Alamitos, pp 418–427

Ramesh B, Jarke M (2001) Toward reference models of requirements traceability. Softw Eng 27(1):58–93

Ran A, Kuusela J (1996) Design decision trees. In: IWSSD '96: Proceedings of the 8th international workshop on software specification and design. IEEE Computer Society, Washington, DC, pp 172–175

Regli WC, Hu X, Atwood M, Sun W (2000) A survey of design rationale systems: approaches, representation, capture and retrieval, engineering with computers, vol 16, Springer-Verlag London Limited, pp 209–235

Russel S, Norvig P (1995): Artifical Intelligence A modern approach. Prentice-Hall, Englewood Cliffs

Tekinerdogan B, Akşit M (2002) Classifying and evaluating architecture design methods. In: Akşit M (eds) Software architecture and component technology. Kluwer Academic Publishers, Dordrecht. pp 3–28

Tretmans J, Wijbrans K, Chaudron M (2001) Software engineering with formal methods: the development of a storm surge barrier control system revisiting seven myths of formal methods, Form. Methods Syst Des 19(2):195–215

Yen J, Lee J (1993) Fuzzy logic as a basis for specifying imprecise requirements. In: Proceedings of 2nd IEEE international conference on fuzzy systems (FUZZ-IEEE'93). IEEE Computer Society press, Alamitos, pp 745–749

Yourdon E, Constantine LL (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice-Hall, Englewood Cliffs

Zadeh LA (1975) The concept of a linguistic variable and its application to approximate reasoning—II. Inf Sci 8(4):301–357